

StormC

The professional choice

Users manual

**ANSI C/C++ Development-
system for the Amiga**



STORMC C/C++ DEVELOPMENT SYSTEM

Software and manual

(c) 1995-1999 HAAGE & PARTNER Computer GmbH

Authors:

Jochen Becher

Editor

Project Manager

Debugger

Profiler

Libraries

Librarian

ScreenManager

Wizard.Library

All rights reserved. This manual and the accompanying software are copyrighted. They may not be reproduced in any form (whether partially or in whole) by any means of procedure, sent, multiplied and/or spread or be translated into another language.

HAAGE & PARTNER assumes no responsibility for damage, caused by or resulting from malfunction of the program, faulty copies or error in the manual are to be led back.

Jens Gelhar

ANSI C Compiler

C++ Compiler

PPC-Frontend

pOS-Compiler

Michael Rock

Optimizing Linker

Patcher

FD2PRAGMA

PPC-Backend

Copyrights and trademarks:

Markus Nerding

Jeroen T. Vermeulen

Wouter van Oortmerssen

Peter-Frans Hollants

Georges Goncalves

Kersten Emmrich

Manual Translation

Amiga is a registered trademark of its owner.

Amiga, AmigaDOS, Kickstart and Workbench are trademarks.

SAS and SAS / C are registered trademarks of the SAS Institute Inc.

The designation of products which are not from the HAAGE & PARTNER COMPUTER GmbH serves information purposes exclusively and presents no trademark abuse.

Peter (dreamy) Traskalik

Hartwig Haage

Graphics



LICENSEE AGREEMENT

1 In general

- (1) Object of this contract is the use of computer programs from the HAAGE & PARTNER COMPUTER GmbH, including the manual as well as other pertinent, written material, subsequently summed up as the product.
- (2) The HAAGE & PARTNER COMPUTER GmbH and/or the licensee indicated in the product are owners of all rights of the products and the trademarks.

2 Right of usufruct

- (1) The buyer does receive a non-transferable, non-exclusive right, to use the acquired product on a single computer.
- (2) In addition the user may produce one copy for security only.
- (3) The buyer is not allowed, to expel the acquired product, to rent, to offer sub-licenses or in any other ways to put it at the disposal of other persons.
- (4) It is forbidden to change the product, to modify or to re-assemble it. This prohibition includes translating, changing, re-engineering and re-use of parts.

3 Warranty

- (1) The HAAGE & PARTNER COMPUTER GmbH guarantees that up to the point in time of delivery, the data carriers are physically free of material and manufacturing defects and the product can be used as described in the documentation.
- (2) Defects of the delivered product are removed by the supplier within a warranty period of six months from delivery. This happens through free replacement or in the form of an update, at the discretion of the supplier.
- (3) The HAAGE & PARTNER COMPUTER GmbH does not guarantee that the product is suitable for the task anticipated by the customer. The HAAGE & PARTNER COMPUTER GmbH does not take any responsibility for any damage that may be caused.
- (4) The user is aware that under the present state of technology it is not possible to manufacture faultless software.

4 Other

- (1) In this contract all rights and responsibilities of the contracting parties are regulated. Other agreements do not exist. Changes are only accepted in written form and in reference to this contract and have to be signed by both parties.
- (2) The jurisdiction for all quarrels over this contract is the court responsible at the seat of HAAGE & PARTNER COMPUTER GmbH
- (3) If any single clause of these conditions should be at odds with the law or lose its lawfulness through a later circumstance, or should a gap in these conditions appear, the unaffected terms will remain in effect. In lieu of an ineffective term of the contract or for the completion of the gap an appropriate agreement should be formulated which best approximates within the bounds of the law the one that the contracting parties had in mind as they agreed on this contract.



PREFACE

- (4) Any violation of this licence agreement or of copyright and trademark rights will be prosecuted under civil law.
- (5) The installation of the software constitutes an agreement with these license conditions.
- (6) If you should not agree with this license agreement you have to return the product to the supplier immediately.

June 1996



PREFACE

“We develop to PowerUp the AMIGA.”

Here it is at last - a new compiler system that gives you the ability to develop powerful applications for the Amiga more easily, efficient and very fast. It is a completely new development system that gives you the tools you have been missing on the Amiga for a long time. New concepts and a look forwards into the future.

Some a 18 months ago we asked ourselves: why is there no development system for the Amiga that beats those of other platforms such as CodeWarrior (Macintosh) or BorlandC++. These programs make it so easy to create good programs. So we started to look for good people who could realise this vision. After a short time we found them:

Jens Gelhar: He did the first C++ compiler for the Amiga in 1992. Now he puts all his experience into the StormC compiler.

Jochen Becher: He is one of the founders of HAAGE & PARTNER. He did his first compiler some years ago. Then he programmed a source-level debugger for a C++ compiler and one of the first C++ Class Libraries for the Amiga.

He is the Project Manager of StormC and he is the father of StormShell and the project manager.

Michael Rock: He too has been programming on the Amiga for a long time. He is responsible for the very fast and very compatible StormLINK - StormC's linker. Now he is working on the PowerPC code generation as well.

Besides these guys many others did a good job of supporting StormC during development. There are assistant programmers, beta testers, many programmers using the demo version and reporting their wishes to us and the customers who always encouraged us to do a little more ;-)

We released the International version of StormC to make its power available to every Amiga programmer. Now it is up to you to “PowerUp the AMIGA” with your fine programs.

We are now working on the next step towards the future: the PPC version of StormC for the new PowerAMIGA.

Now we want to thank our beta testers and all the people who supported us during the development of StormC.

Thank you very much:

Olaf Barthel
Holger Burkarth
Thomas Bayen
Bernhard Büchter
Jan-Claas Dirks
Mario Kettenberger
Alexander Pratsch
Michael Rock
Jürgen Schildmann
Stephan Schüerholz
Thomas Wilhelmi
Heinz Wrobel

Special thanks to Jeroen T. Vermeulen.

Particular thanks goes to Gudrun Volkwein, Bernadette Becher and Hartwig Haage.



TECHNICAL SUPPORT

In case of problems concerning StormC you should:

1. Check your installation for completeness and read the ReadMe file.
2. Check the installation of your operating system for completeness and verify that all relevant parts contain the right version numbers (at least OS 3.0).
3. Check programs that are running in the background. There might be some software running on your Amiga that will interfere with StormC. In particular some tools and patches, which are loaded in the Startup sequence, can considerably affect the mode of operation. Try starting your Amiga without these programs, to be sure that they are not the cause of the trouble.
4. Please keep your Registration number handy.
5. Write down the version number of StormC ("About" window) and the version numbers and build dates of components (Storm.Library, StormC, StormCPP, StormEd, StormLink, StormRun, StormShell). You get them by typing the command

version "file name" full

in CLI or Shell.

6. Please note your hardware and software configuration too.
7. If you think that your problem is caused by an error of StormC then please try to narrow down the error to the smallest possible piece of your code and send it to us (by mail or e-mail).

You can tell us your problems through many channels:

Internet: storm-support@haage-partner.com

Contacting us by e-mail is most convenient to handle for us. We can forward your problems to the developers very easily. If you could send us a code segment containing the error, this would certainly speed up the process of fixing the

bug tremendously. Via e-mail we can respond to your message very easily or send you an individual patch or advice. Please use e-mail if at all possible.

WWW homepage: <http://www.haage-partner.com>

On our homepage on WWW on Internet you will find the current information on StormC. Here we also have a special support area with hints, patches, bug fixes and a lot of information about StormC.

HAAGE & PARTNER Computer GmbH
Schlossborner Weg 7
61479 Glashuetten
Germany

Tel: ++49 - 61 74 - 96 61 28
Fax: ++49 - 61 74 - 96 61 01

The Hotline is occupied from Monday through Friday from 3:00pm to 7:00pm o'clock. Please be prepared with the information the support-staff members will need. This will speed up the solution of your problems.

We prefer Support in writing because some problems can't be solved by phone, so we recommend the use of e-mail, fax or normal mail.



COPYRIGHT 2

STORMC C/C++ DEVELOPMENT SYSTEM 2

Licensee agreement 3

PREFACE 4

Preface 5

"We develop to PowerUp the AMIGA." 5

Technical support 7

1 WELCOME 19

AMIGA IS BACK FOR FUTURE. 19

If you don't like manuals 19

If you are familiar with C and C++ 19

If you never worked with a compiler nor did any programming in C 20

2 INSTALLATION 21

OVERVIEW 22

Before you install 22

Localization 22

Installing with low memory 23

Full installation 24

Novice User: 24

Intermediate User: 24

Expert User: 24

Selecting the installation directory 25

Installing an update 26

Removing StormC 26

Custom files in the StormC drawer 27

After Installation 27

Troubleshooting the Installation 27

Correct StormC installation 27



3 FIRST STEPS 29

BEFORE YOU BEGIN 30

Running the program 30

StormShell 30

StormC 31

StormLink 31

StormEd 31

StormRun 31

StormASM 31

Toolbar access 32

Keyboard control 32

The icon bar offers the following functions: 32

THE CONCEPT OF PROJECTS 33

The simple way 33

An easier approach through batch files 33

STORMC'S PROJECT MANAGEMENT 34

Creating a new project 34

What is a project? 35

Make and module dependencies 35

Saving the project and creating a new directory 35

Adding files to the project 37

Automatic Usage of Preferences 37

Specifying the program's name 38

Saving the project 38

Creating a source file 39

Adjusting settings 40

Compiling the source code 40

Running the translated program 42

Console output 42

4 PROJECT MANAGER 45

OVERVIEW 47

Some words about the startup 47

**Tooltypes 47***The ScreenManager 48***Memory usage 51****Settings 51***Automatic storage 52**Automatic storage of unnamed files 53**Automatic backup copy 53**Selecting text font 53***ORGANISATION OF A PROJECT 55****Creation of a new project 55****Setup of projects 56****Project Sections 56****Adding Files 57***Adding Sources 58**Add Window 58**Adding Libraries 58**Choosing a program name 59***Drag&Drop 59****Controls of the Project Window 59***Folding of Sections 60**Showing Sources 60**Starting a Program 60**Open a Child Project 61***Keyboard control 61***Cursor keys 61**Return 61***Deleting Items of a Project 61****Organising files of a project 61***Project specific headers 62**Documentation, scripts ... 63***Makescripts 63****Passing arguments to makescripts 66***Assembler scripts 67***Saving of the paths 69****PROJECT SETTINGS 70****Paths and global settings 70***Include path 71*

Header Files 71

Working memory - Workspace 72

Definitions and Warnings 72

ANSI-C/C++ Settings 74

Source 74

Template Functions 74

Exceptions 75

Debugger 76

Code Generation 76

Processor Specific Code Generation 77

Quality of the Optimisation 78

Optimising 78

Compiler Warnings 80

Standard of the language 80

Security 81

Optimisation 81

Path Settings and Linker Modes 82

Generation of Programs 82

Library Path 84

Warnings 84

Optimiser 84

Hunk Optimisations and Memory Settings 85

Summarise Hunk 85

Manner of Memory 86

ROM Code 86

Call of the executable program 87

Execute Environment 87

Start From CLI 87

I/O Settings 88

Input/Output 89

Save Project 89

5 STORMED 91

IN GENERAL 93

New text 93

Controls of windows 93

Open / Load Text 94

Tooltips 94

**Save Text / Save Text As ... 94****Free Text 94****Options of Text 95***Tabulations and Indents 95**Indent ahead and behind of brackets 96**Dictionaries and Syntax 97**Dictionaries 97**Syntax 98**Colour Settings 99**File Saving Settings 99**Saving the Settings 100***Keyboard Navigation 100***Cursor-Keys 100**<Return>, <Enter>, <Tab> 100***Undo / Redo 100****Block Operations 101***Mark, Cut, Copy and Paste 101***The Mouse 102****Find and Replace 102***Direction 103**Mode 103**Ignore Upper/Lower Case 103**Ignore Accent 103**Find 103**Replace 104**Replace All 104*

6 COMPILER 105

SPECIAL FEATURES OF STORMC 107**DATA in Registers 107****Parameters in Registers 108****Inline Functions 108****The Pragma instructions 110***Data in Chip and Fast RAM 110**AmigaOS Calls 111**The #pragma tagcall 111***The #pragma priority 112***Constructors and Destructors in ANSI C 113*

Constructors and Destructors in C++ 113

Priority list 113

Joining Lines 114

Predefined symbols 115

Build your own INIT_ and EXIT_ routines 117

Use of Shared libraries 117

Prototypes 117

Stub Functions 118

#pragma amicall 118

Forced opening of a Amiga library 122

PROGRAMMING OF SHARED LIBRARIES 123

The Setup of a Shared Library 123

The #pragma Libbase 124

Register Setup 124

Shared Library Project Settings 125

The setup of FD files 125

The first four Functions 126

Home-made Initialisation and Release 127

Important hints to Shared libraries 128

PORTING FROM SAS/C TO STORMC 130

Project settings 130

Syntax 131

Keywords 131

CLI VERSION OF THE COMPILER 134

The instruction 134

Options 134

Assembler source 136

Pre-processor: Definition of symbols 136

Pre-processor: Include files 137

Compiler mode 137

ANSI C or C++ 137

Exception handling 137

Creation of Template functions 138

Code creation 138

Data model 138

Code model 138



Optimisations 138
Code for special processor 140
Code for linker libraries 141

Debugger 141

RunShell 141
Symbolic debugger 141

Copyrights 141

Warnings and errors 141

Format of the error output 141
Colours and styles 142
Error file 142
Optional warnings 143
Treat warnings like errors 144
Core memories 144
Pre-compiled header files 144

Summary 145

7 THE DEBUGGER 149

GENERAL INFORMATION ON RUNSHELL 151

The StormC Maxim 151
A Resource-Tracking Example 152
Freeze the program temporarily 153
Halting the program 154
Changing the task priority 154
Sending signals 154

USING THE DEBUGGER 157

The Variable Window 159
Temporary Casts 161
Changing Values 161
Sorting of Variables 161

THE PROFILER 164

Profiler technical information 167

REFERENCE 169

Control Window 169

Status line 169

"Program Stops At Breakpoint": 169

"Continue Program": 169

"Program waits for ...": 169

Debugger icons 169

Go to next breakpoint 170

Step in (single step) 170

Step over (single step, but execute function calls without stopping) 171

Go to the end of the function. 171

Show Current Program Position 171

Pause 172

Kill 172

Priority gadgets 172

Signal group 172

Protocol gadgets 172

Window close gadget 173

Current variable window 174

The module window 177

The function window 178

The history window 178

The breakpoint window 179

The address requester 180

The hex editor 181

Choosing the display 181

The address string gadget 181

The address column 181

The hexadecimal column 181

The ASCII column 181

Keyboard control 182

The scrollbar in the hex editor 182

8 THE LINKER 183

THE LINKER,

THE UNKNOWN CREATURE 185

A first example 185

ANSI-C Hello World 185

Startup Code 186

**Usage 189****Parameters 189****Memory classes 191****Compatibility 200****Error Messages 200****Error Messages 201**

Unknown symbol type 201

16-bit Data reloc out of range 201

8-bit data reloc out of range 201

Hunk type not Code/Data/BSS 201

16-bit code reloc out of range 201

8-bit code reloc out of range 201

Offset to data object is not 16-bit 201

Offset to code object is not 16-bit 202

Offset to data object is not 8-bit 202

Offset to code object is not 8-bit 202

Data- access to code 202

Code- access to data 202

InitModules() not used, but not empty 202

CleanupModules() not used, but not empty 203

File not found 203

Unknown number format 203

Symbol is not defined in this file 203

Nothing loaded, thus no linking 203

Can not write file 203

Program is already linked 204

Overlays not supported 204

Hunk is unknown 204

Program does not contain any code 204

Symbol not defined 204

Symbol renamed to _stub 204

_stub is undefined 205

32-Bit Reference for Symbol 205

32-bit Reloc to Data 205

32-bit Reloc to BSS 205

32-bit Reloc to Code 205

32 Bit Reference for symbol from FROMFILE to TOFILE 205

Jump chain across hunk > 32 KByte not possible 205

More than 32 KByte merged hunks 205

Illegal access to Linker-defined Symbol 205

Fatal errors : aborting 206

Hunk_lib inside Library ?? 206

Hunk_Lib not found 206

Wrong type in Library 206

Predefined values 206

Symbols for data-data relocation 208

Hunk Layout 208

Memory Classes 208

Order of Searching for Symbols 209

Hunk Order 209

Near Code / Near Data 209



e thank you for buying this development system. Be assured that this was the right decision. StormC is the best tool to speed up your software development, to make it more comfortable and easy.

With StormC we give you a tool for the future of the Amiga. With the help of StormC you can build applications of outstanding quality that help assure the persistence of the Amiga.

After working a while with StormC you will never understand how you ever did without it. We put all our experience into it and development is still going on. Next step is the PowerPC version for the PowerAmiga. Then you will be one of the first who's applications will run in PPC native code. And then

AMIGA IS BACK FOR FUTURE.

If you don't like manuals

Are you somebody who never reads a manual? So working with computers must be inborn. Congratulations!

In most cases there will be a problem that you can not solve on your own. You could phone us to get the answer, but you can also have a look at the manual. If you are a beginner you should work through the basics. Read the first chapters about the editor, project manager, and compiler. Use the rest as a reference and work through it if you need to. If you still have questions, you can contact us then. This will work best for all of us. :)

If you are familiar with C and C++

In this case you should have a look at the concepts of StormC. You will only will get the best out of you programming system by knowing the details.



If you never worked with a compiler nor did any programming in C

Then StormC is best for you. One of our main goals was to design a development system that makes programming easier. So we did an integrated system where you don't need to care about the individual parts. You simply type your source into the editor and then hit "Run" and the rest will be done by StormC: saving, compiling, linking, running. We recommend that you try out "First Steps". There you will get the basics of StormC. Later you should work through the first chapters of the manual: editor, project manager, compiler. Later you should read through the other chapters as well to get an overview of the possibilities of StormC.



This chapter describes how to install StormC on your hard disk. To avoid installation problems you should proceed the installation step by step as described here.

OVERVIEW	22
Before you install	22
<i>Localization</i>	22
Installing with low memory	23
Full installation	24
<i>Novice User:</i>	24
<i>Intermediate User:</i>	24
<i>Expert User:</i>	24
Selecting the installation directory	25
Installing an update	26
Removing StormC	26
<i>Custom files in the StormC drawer</i>	27
<i>After Installation</i>	27
<i>Troubleshooting the Installation</i>	27
<i>Correct StormC installation</i>	27

OVERVIEW

To guide you through the installation, Commodore's Installer will be used. It has become the standard application for this purpose. You should have no problems using it.

Before you install

StormC is shipped on a set of disks or a CD. Please put the first disk into your disk drive or the CD into your CD-ROM drive.

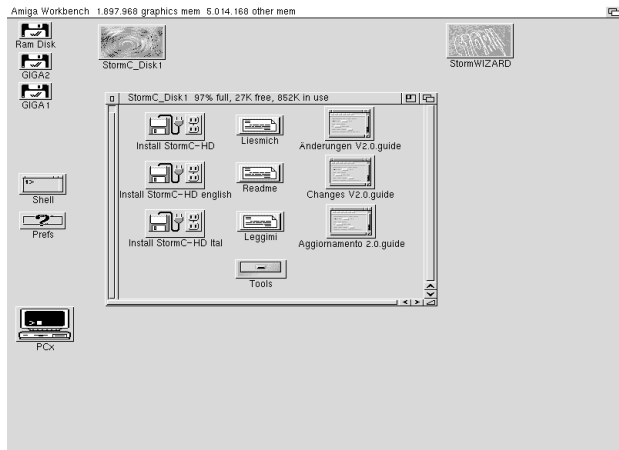


You can find information on the memory

usage of the system in a file called "**Readme**" on the disk. This file also contains information about recent changes since the manual was printed.

Before you install make sure that there's enough space on your hard disk for StormC. Otherwise the installation can be interrupted.

After you insert the disk or the CD, double-click the drive's icon, and a window similar to the following illustration is shown on your monitor.



Localization

Depending on which language you prefer you can decide whether the installation should occur in the German or English language. This setting does not affect StormC's localization settings. Both choices offer you the possibility to install one or more localization catalogues.



Installing with low memory

Even if there's only a little space left on your hard disk you can still install StormC. In this case run the Installer in Expert Mode. You will then be asked to confirm the installation of every part of the package.

The following components are required for StormC and should be installed:

Include (dir)

- all Includes except for the "**Proto**" and "**Pragmas**"

drawers Lib (dir)

- Storm.lib
- Amiga.lib

StormSYS (dir)

- Appmanager.library
- StormC
- StormCPP
- StormEd
- StormEdGUI.wizard
- StormLink
- StormShell
- StormShellGUI.wizard
- (StormRun)
- (StormRunGUI.wizard)

LIBS: (dir)

- wizard.library

You don't necessarily need to install the run time system and the debugger. However this also gives you the possibility to run and debug your programs right out of the compiler environment.

You may of course leave off the entire environment and proceed with an even smaller installation. The StormSYS drawer should then contain these programs:

StormSYS (dir)

- StormC
- StormLink

Starting the installation

Double-click one of the icons. Soon the installer's window will open.



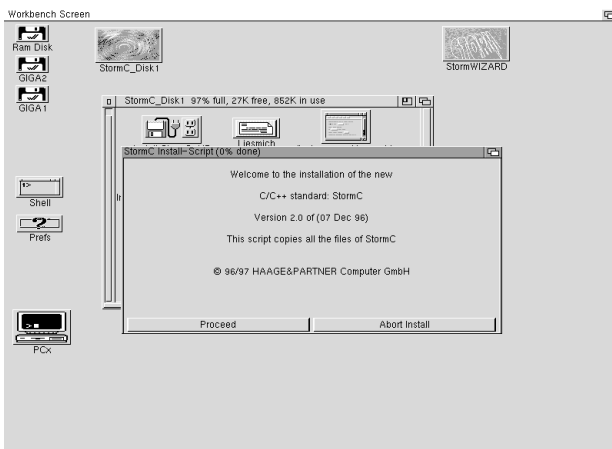
Normal Installer-Icon



Selected Installer-Icon

Full installation

After starting, the installer's window appears with information about its version number and date. At the bottom you can see two gadgets. Click on "**Continue**" and the next page of the installation will appear where you can choose settings concerning the installation process. A click on "**Abort installation**" and answering "**Yes**" at the "**Are you sure?**" prompt will quit the installation. These two gadgets will guide you on every page of the installation process.



The second page allows you to set parameters affecting the entire installation process.



If there is already an old installation on

your Amiga it will be replaced in novice mode without further information.

Novice User:

In this mode everything is automated. After selecting the drive to install to a complete installation will be performed. You will only be asked to change disks, if necessary.

Intermediate User:

This mode gives you more control over the installation. The installer detects a previously existing installation and asks you if you want to reinstall from scratch, install an update or remove the old installation.

Expert User:

The expert mode gives you nearly full control over the installation. This is useful if you don't want to install the full package but just parts of it.



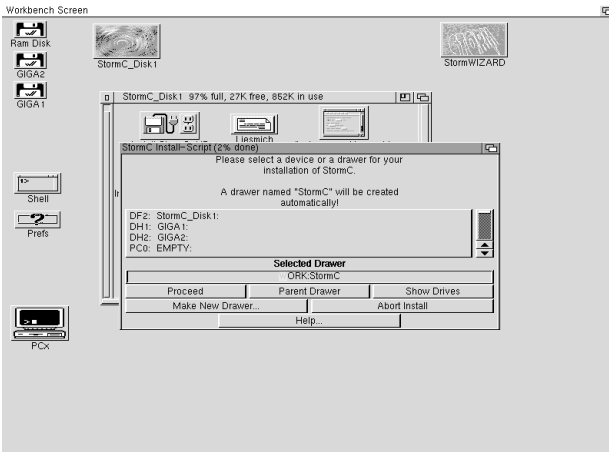
Each time the installer is about to copy something you will be shown a list of files. Only the files you select will actually be copied. Furthermore this page allows you to change the destination path. This is, however, not advised, as it makes little sense. The complete installation, except for some tools and configuration files, will be done in a single directory ("**Drive:StormC**") and should not be spread around manually to multiple directories.

This prompt only exists in the "Intermediate User" and "Expert User" modes, and only if there's already an existing installation, including the first StormC Preview, and the current demo versions. For more information refer to "**Installing an Update**" and "**Removing StormC**".

Selecting the installation directory

When installing from scratch you select a drive or directory where the StormC file will be stored.

 The installer will create a directory "StormC" on the selected drive or in the selected path. Diskette drives and the RAM disk are not allowed as destinations.



Click on "**Show drives**" if you don't want to install on the currently displayed drive, which is the default destination. Then click on a drive in the list shown.

Select "**Create a new drawer**" if you wish to create a new drawer inside the directory currently shown. However, keep in mind that the installer will automatically create a "StormC" drawer and copy all files there.

"**Parent drawer**" takes you one level back in the directory structure.

 After selecting the destination path, the file "**Readme**" will be copied and displayed. You should read about recent changes since the manual was printed during the installation process.

Select **"Continue"** when the installation location is okay.

Depending on the user level chosen, the installation is performed automatically. We do not recommend changing the destination for parts of the package.



During an update, all files you modified (e.g. includes, demos and preferences) will be overwritten. You should therefore create a backup of your changes prior to updating and restore it afterwards.

Installing an update

The installer automatically recognises if there's already an existing version of StormC on your hard disk. It will notify you (see the illustration on the previous page) and offer to reinstall from scratch, install an update or remove the old installation completely.

When updating, you, of course, won't be asked where to install since the old location is already known.

In "Novice User" mode the old installation will be overwritten without further inquiry.

In "Intermediate User" mode the installer will check before unpacking whether the current file already exists and prompt whether it should be overwritten. With parts consisting of multiple files the installer will not ask to replace each file.

The "Expert User" mode guarantees full control over the Update installation. Files will be temporarily unpacked into RAM, and you will then be asked for every single file to be copied.

Removing StormC

We regret if you wish to remove your StormC installation from your hard disk, but it will be mentioned here anyway.



Only files copied during the StormC installation will be removed.

You will, of course, only have the possibility to remove the installation if the installer detects it.

Depending on the User level chosen, you will be asked for each file whether or not you want to remove it.

As only a few files are required outside the StormC drawer, a manual deinstallation should be no problem. The icons for the editor files and project management are removed from the **"ENVARC:"** and **"ENV:"** directories. The StormScreen-Manager is removed from your hard disk's **"WBStartup"** drawer. Of course, the Assignment in your **"User-Startup"**



will be removed, otherwise you would get an error message during the next reboot of your system, that the drawer does not exist anymore.

Custom files in the StormC drawer

Of course, StormC will not be removed if the Installer detects custom files in it. You will be notified of this and need to move these files somewhere else on your harddisk and delete the "**StormC**" drawer yourself.

After Installation

If you did not fill out the registration card yet this is the best time to do so. You should by all means register so that we can inform you regularly about updates and upgrades.

Troubleshooting the Installation

If you have followed the listed instructions, you should have no problems during the installation. It is, however, impossible to predict when a file or even a complete disk is damaged. If only one, or just a few files are damaged, the installation may be performed manually.

Please inform us as soon as possible about the damage. We will then send you a replacement disk.

To allow you to continue working with the compiler system the following list gives you information about the files and their required location.

Correct StormC installation

On the first StormC disk you can find a list of the files of a complete StormC installation.



2 *INSTALLATION*



Every C-compiler's introduction begins with the "Hello World" program, so will we.

With this simple example we want to show you the first steps you are waiting for. You will learn how to create a simple project, to write new source, to run the compiler, and, finally, to run the program.

BEFORE YOU BEGIN	30
Running the program	30
Toolbar access	32
<i>Keyboard control</i>	32
<i>The icon bar offers the following functions:</i>	32
THE CONCEPT OF PROJECTS	33
The simple way	33
An easier approach through batch files	33
STORMC'S PROJECT MANAGEMENT	34
Creating a new project	34
What is a project?	35
Make and module dependencies	35
Saving the project and creating a new directory	35
Adding files to the project	37
Automatic Usage of Preferences	37
Specifying the program's name	38
Saving the project	38
Creating a source file	39
Adjusting settings	40
Compiling the source code	40
Running the translated program	42
Console output	42

BEFORE YOU BEGIN

The following instructions assume that you have successfully installed StormC and rebooted your system. If you didn't, please go back and read Chapter 2: Installation.

Running the program

Please start StormC with a double-click on its icon (see the marginal illustration). You can find the icon in the drawer in which you installed StormC.



After starting, you will see a Welcome message which remains until all of StormC's components are loaded. If your Workbench has more than 32 colours you will see a Welcome picture as shown below.



The following programs, which belong to the compiler environment, will be run:

StormShell

The StormShell unifies the different parts of the compiler environment. It is responsible for important jobs such as project management, without which programs consisting of multiple modules would not be possible.



StormC

The compiler is the heart of the StormC developing environment. The compiler accepts both ANSI-C and C++ sources and performs optimisations as well. It supports current ANSI standards and all features of the AT&T version 3.0 compiler systems (CFRONT).

StormLink

This amazingly fast linker is the interface to other compiler systems. It is able to process link libraries from the SAS/C and MaxonC++ compiler environments. This feature allows you to use all common libraries in your programs.

StormEd

We are sure that you will soon appreciate our editor's two major features. Colorization of strings, comments, constants etc. helps you to quickly and efficiently locate and remove bugs. The included lexicon covers all functions and structure names used on the Amiga. Because of the colorization you will notice immediately whether an entered function name is correct.

Another feature is the Editor's usage for debugging purposes. When you start the debugger, already open windows and functions will be reused. You can continue working with the Editor as usual. Only the breakpoint column at the left and the brightly rendered display of the program counter will remind you that you're currently debugging your program.

StormRun

StormRun contains the Source-Level-Debugger and the code necessary to allow your programs to run from within the development environment. It is also responsible for Resource Tracking.

StormASM

StormASM is not a complete Assembler, but an interface to an external Assembler, PhxAss. Its demo version and documentation are included.

StormC's major controls will be displayed as soon as it is loaded. The icons allow for access to the compiler's most important functions.



Due to the integration in the compiler environment you won't notice that it consists of multiple programs. The communication between the parts occurs using an ARexx port which, as another benefit, offers very flexible interfaces to other programs.



Some "Sun-Mouse tools" may cause problems when the help line is rendered. To fix this bug we have supplied the program "MagicMenuFix". During the installation you will be asked whether you wish to install it in your WBStartup drawer. Refer to the "Readme" file for more information.

Toolbar access

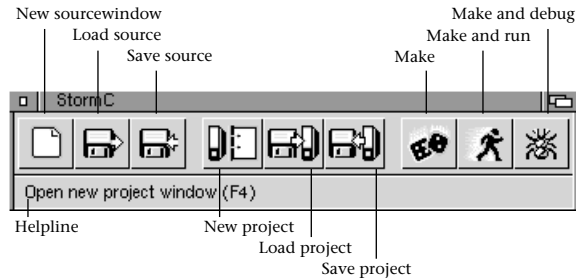
Their accessibility through the icon window makes these functions instantly available. A single click on an icon is enough to cause the associated function to be executed. As you have certainly already noticed a help line gives you details about these functions.

This feature is particularly important when you are just beginning to use StormC. You will soon ignore the help line more often and remember the functions associated with the icons. You will find the help also in other parts of the environment where icons are used.

Keyboard control

The icon's functions are also accessible using the function keys. <F1>-<F3> are used for the text functions, <F4>-<F6> for the project functions and <F8>-<F10> for the compiler and debugger functions.

The icon bar offers the following functions:





THE CONCEPT OF PROJECTS

Before we start writing and compiling source code I'd like to discuss some basic aspects of writing programs.

The simple way

With a "traditional" compiler system you would start a text editor, enter the source code, save it and run the compiler. Additionally, you would perhaps also specify extra options, or use the default settings, perhaps kept in an environment variable.

The compiler would then create an object file, and you would run the Linker to get an Executable. Finished!

An easier approach through batch files

It would be easier if you created a batch file containing the commands you previously entered manually. After a change to the source code you'd then only need to run the batch file.

This ease would be clearly perceptible, which results in higher turn-around times and thus in a higher programming efficiency.

This approach has no disadvantages as long as you're only working with a single source file. As soon as you integrate two or more source files into the batch file, every time you call it you would have all source files compiled - regardless of whether a source file has been changed or not.

Thus, the turn-around times would always stay the same. With some little expense it would be possible to check whether a source file is newer than its associated object file but that would be all you could do in a justifiable amount of time.

Another major aid in this case is a so-called Make program which only passes modified source files to the compiler and also supports dependencies. For details about Make refer to chapter 4 for a more intensive discussion.



StormC basically works with projects. Source

files can only be compiled when they're part of a project. Read the following introduction step by step and you will learn about the simple usage of projects.

STORMC'S PROJECT MANAGEMENT

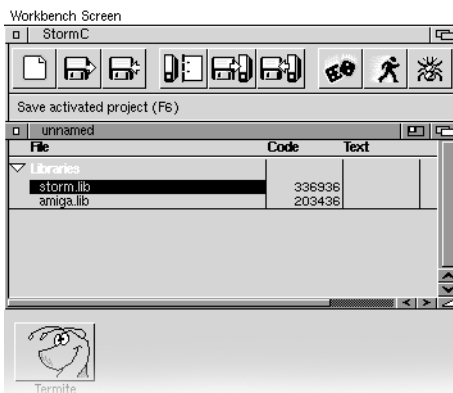
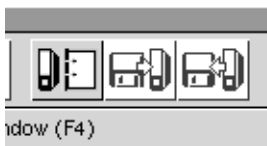
Similar to batch files, a StormC project contains a list of files and settings. This is, however, created visually, which makes it easier to use and to understand. A StormC projects unifies all parts belonging to a program so that they can be managed centrally. This includes files which do not don't directly affect the creation of a program, such as AmigaGuide documents, and ARexx and Installer scripts.

The project manager has similar properties to a Make program and reacts accordingly when multiple source files are used. Dependencies between source files and header files are also no problem.

Creating a new project

The most important step to compile a source file using StormC is to create a new project. Source files can't be compiled without being associated with a project. This may appear to be a limit at first, but will save you a lot of trouble later on. Creating a project is really simple and helps to keep your programming efforts ordered.

Please click on "**New project**" in the icon bar. A new project window will be displayed.





What is a project?

A project unifies everything belonging to your program: C, C++ and Assembler source files, header and object files, link libraries, documentation, graphics, pictures and other resources. The collection remains easy to manage through separation into different sections. The project manager is, however, also a visually orientated Make.

Make and module dependencies

At every compiler run, the dependencies between ".o", ".h", ".ass", ".asm", ".i" and ".c" files (of course also ".cc" or ".cpp") will be evaluated and passed to the project manager. This allows the project manager to know that a C source file needs to be recompiled when a ".h" header file has changed that is being included in the ".c" source.

When you click on "**Make**", "**Run**" or "**Debug**" all dependencies will be checked, and Make decides which program modules need to be recompiled and which do not. The only difference between "**Run**" and "**Make**" is that "**Run**," after successful compilation, automatically runs the program. Selecting "**Debug**" will automatically start the Debugger after compilation.

Saving the project and creating a new directory

Before starting the next step, please save the project and create a new directory for it.

Please click on the "**Save project**" icon. In the displayed standard ASL file requester please choose the "**StormC:**" directory and enter as file name "**Hello World/Hello world**".



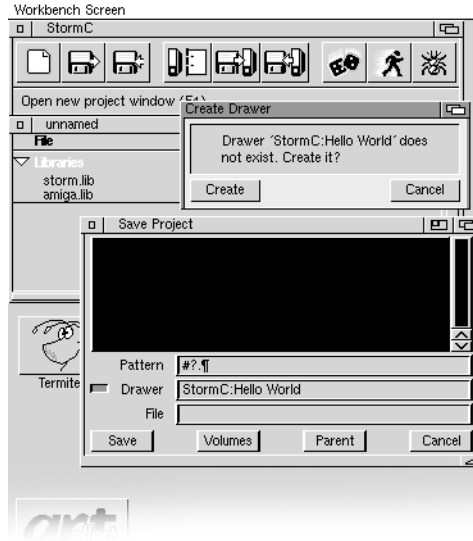
*The illustration next to this text shows a new project created by a simple click on the "**New project**" icon. The project's preferences can be modified freely and saved as default settings for new projects.*

3 FIRST STEPS



The ".?" suffix can also be entered manually by pressing <ALT>+<P>.

The ".?" suffix is automatically appended and signifies that this file is a StormC project.



When you have entered the file name as shown above into the text gadget and pressed <Return>, you will be asked if you want to create a new drawer. In order to keep a better overview you should create a new directory for every project and proceed as shown.

After confirmation of the **"create a new drawer"** requester, you can click the **"OK"** gadget in the ASL file requester and your project will be saved in the new directory.

You may wonder why the empty project needs to be saved; even if the project is empty it is recommended to specify a clear project path at the start. The names of the source files and other resources can then be integrated and saved relative to the project's path (otherwise the absolute path would be used). Another advantage is that when adding new files to the project the ASL file requester already contains the project's path, so you won't have to enter it manually.

Adding files to the project

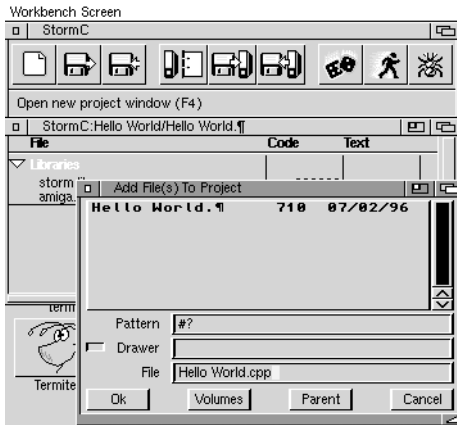
To continue, please choose "**Add file(s)**" from the "**Project**" menu. Then enter "**Hello World.cpp**" as the filename. The suffix should be usable to distinguish between ANSI-C, C++ and Assembler source files and header files. In our example we chose **".cpp"** because we want to write a "**Hello World**" example in C++.



When adding empty source files created with "Add file(s)" the appropriate preferences for the given suffix will be taken from the preference icons located in the ENV: location.



This illustration shows the ASL filerequester where existing files may be selected and new files can be created. Please activate the "File" text entry field with the mouse and enter "Hello World.cpp" as file name.



Automatic Usage of Preferences

When you create a new file for use in the project, the file's preferences will be taken from the preference icons according to the specified suffix.

In our example the properties for the new source file "**Hello World.cpp**" are taken from the preference icon "**ENV:STORMC/def_text.cpp.info**". For files ending in **".c"** the properties would be taken from "**ENV:STORMC/def_text.c.info**" etc.

Of course you can modify and set the preferences to your needs. The icon's tooltypes contain all settings made in the "**Editor**" and "**Text**" sections. You can access these in the "**Settings**" menu.

To save the modified settings use **"Save As Default Icon"** in the **"Project"** menu. You will be asked whether you wish to save the settings to the icon with the settings for the appropriate suffix or to **"ENV:STORMC/def_text.info"**, which is responsible for the global settings.



Specifying the program's name

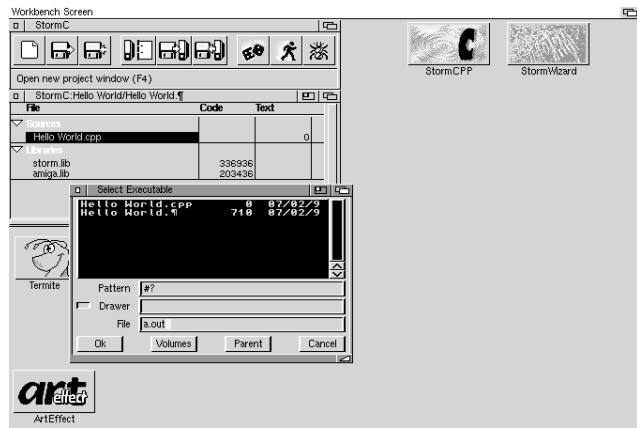
We will now specify the program name to be used by the Linker. If a project does not contain one it will create a program named **"a.out"** - if you link it at all.



When using the Debugger additional files

ending in **".debug"** will be created for every source file. In order to inform the Debugger of which modules the program consists of, a file ending in **".link"** will be created. If you have chosen a different path for the program than the project path, this **".link"** file will be created there, too.

Please choose **"Select Executable..."** from the **"Project"** menu.



The displayed file requester will show the contents of the project's path. You can enter a file name in this path or change it to suit your needs.

Saving the project

Next is an introduction to the text editor. Before proceeding, you should save the project again. Since the last time you saved, you have followed the steps **"Adding files to the project"** including creating a new drawer and **"Specifying**

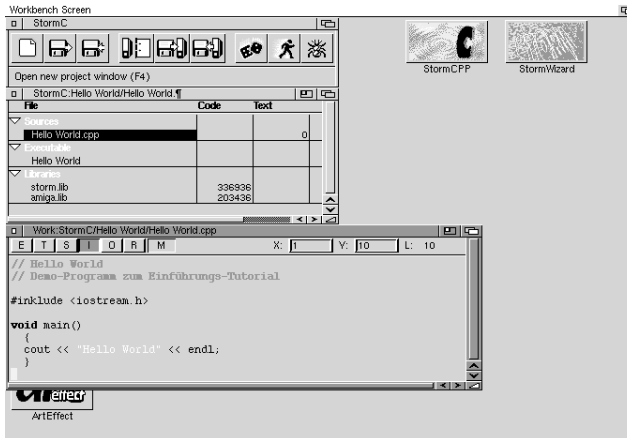
the program's name". Though it would not be very tragic, it would be annoying if these steps were destroyed by something unpredictable such as power failures etc.

To save the project just click on **"Save project"** or press the function key <F6>.

Creating a source file

We will now proceed to the real programming. Open the previously created source file by double-clicking its name in the project.

An empty editor window with the title **"Hello world.cpp"** will appear. Before you begin to enter text please have a quick look at the editor's controls. This illustration may help you.

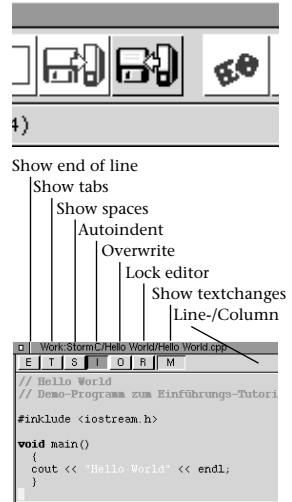


Please enter the following lines:

```
//    Hello World
//    Example program belonging to the
//    introductory tutorial
```

```
#include <iostream.h>
void main()
{
    cout << "Hello World" << endl;
}
```

Save this text by clicking on the **"Save text"** icon in the tool bar.



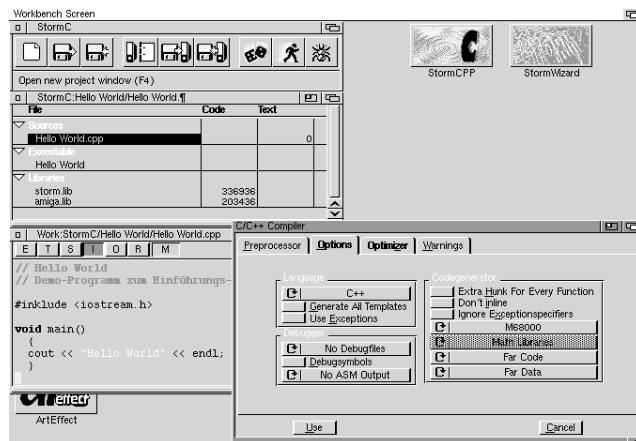
Adjusting settings

Before we compile the program please make sure that the compiler's settings are correct. Please choose the entry **"Compiler..."** out of the **"Settings"** menu. The window found in the illustration below will appear. If you open it for the first time your window will look slightly different than shown below.

Because the project settings could not be placed in a single window we have divided and separated them onto different pages.

At the upper border of the window you can find a paging gadget. With clicks you can browse between the different settings pages.

Please select the **"C/C++ Options"** page.



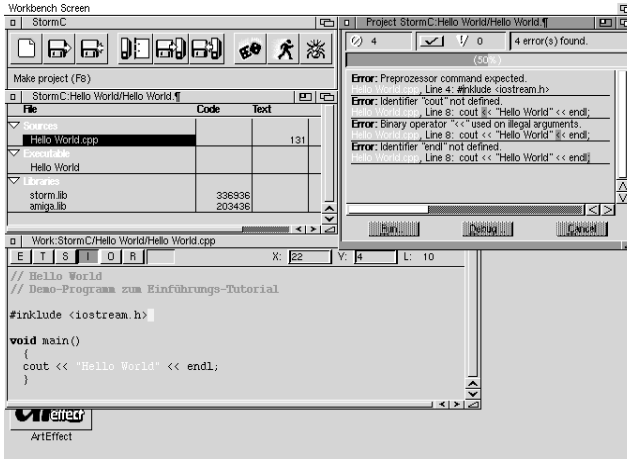
Make sure that the cycle gadget in the **"Language"** group shows **"C++"**. All other settings should be set as shown in the illustration.

Confirm these settings by clicking on **"Use"** or press the <U> key.

Compiling the source code

Clicking **"Make"** will open the error window and the source code will be passed to the Compiler. The compiler and the

linker will output status and progress reports in the error window.



If the compiler finds an error it will be reported in the display below. You will find a very detailed error description, the line number, and the source file where the error was found.

In our example we intentionally made a mistake in line 1. Instead of **#include** we wrote **#inklude**. This is a very silly mistake you should have noticed when you entered the text. If you have colorization enabled in the editor the word **#include** will be rendered in a special colour. As soon as you change only a single character, the color changes to the standard color which is usually black. This allows you to find careless mistakes while entering your text, and avoids unnecessary compiler runs.

To demonstrate the error window, this simple mistake should be included in the source. As soon as the compiler finds the mistake the appropriate error message is shown in the window.

Of course a double-click on the error message is enough to cause the project manager to load the source code and show the error line in the Editor. You can then correct it and run the compiler again.

Running the translated program

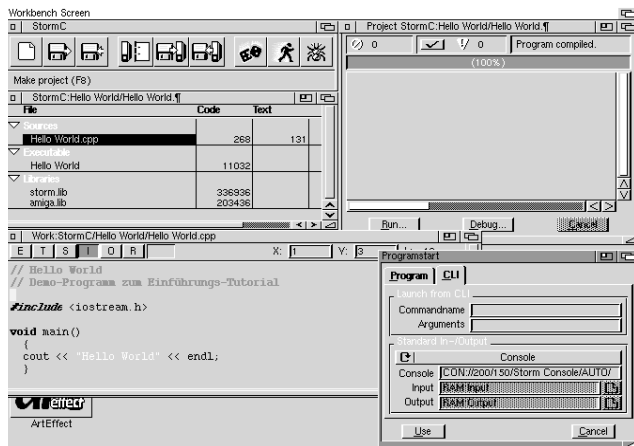
After successful compilation and linking you can click on the **"Run"** button (which was previously ghosted) or the **"Run"** icon to start the program. You can also press the function key <F9> instead.

If you click on the **"Run"** (<F9>) instead of the **"Make"** (<F8>) icon the project manager first checks whether all modules have been compiled. If not, the compiler will be run for all untranslated modules. After successful translation the program will be automatically run.

Console output

If you have already started the program you will have noticed that the program's output could only be seen for a short while, and disappeared almost immediately. The program is very short and thus runs very fast. To see the output window also after the program has finished we need to specify the **"x/y/width/height/title/options"** parameters when opening the console.

Please open the project settings once again by selecting the menu item **"Settings/Program Start..."**.



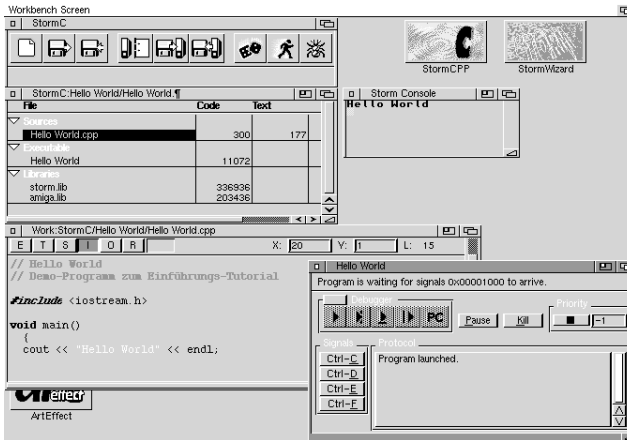


In the second page of the window you can choose settings regarding the in- and output. Please enter the following line in the **"Console"** text field:

CON://320/200/Hello World/CLOSE/AUTO/WAIT

Please restart the program by double-clicking the program's name in the project.

 Double-clicking the program's name in the project is sufficient to re-run the program.



The program will still be executed very fast but keep the console window opened. Close it by clicking the window's close gadget.



Project Manager

4



IBM PCs.

As you have worked through the beginning tutorials, you have heard about the project manager. This is something new to Amiga compiler systems, but it is a common thing on other computer systems such as the Macintosh or

One of the goals of the development environment is to make work easier and more efficient. The project manager is a big step in this direction.

If you look at traditional development environments, you will recognise that every effort was made to build tools around the compiler, because it was the "heart" of the system. Nowadays, this has changed. The "heart" of the system is the project manager. It assembles all parts of a project, and it navigates and controls every part of it.

The following chapter will tell you more about the project manager, how it works, how to configure it, and how to navigate throughout the development system with it.

OVERVIEW	47
Some words about the startup	47
Memory usage	51
Settings	51
ORGANISATION OF A PROJECT	55
Creation of a new project	55
Setup of projects	56
Project Sections	56
Adding Files	57
Adding Sources	58
Add Window	58
Adding Libraries	58
Choosing a program name	59
Drag&Drop	59
Controls of the Project Window	59
Folding of Sections	60
Showing Sources	60
Starting a Program	60
Open a Child Project	61
Keyboard control	61
Cursor keys	61
Return	61

Deleting Items of a Project	61
Organising files of a project	61
Makescripts	63
Passing arguments to makescripts	66
Saving of the paths	69
PROJECT SETTINGS	70
Paths and global settings	70
<i>Include path</i>	<i>71</i>
<i>Header Files</i>	<i>71</i>
<i>Working memory - Workspace</i>	<i>72</i>
Definitions and Warnings	72
ANSI-C/C++ Settings	74
<i>Source</i>	<i>74</i>
<i>Template Functions</i>	<i>74</i>
<i>Exceptions</i>	<i>75</i>
<i>Debugger</i>	<i>76</i>
<i>Code Generation</i>	<i>76</i>
<i>Processor Specific Code Generation</i>	<i>77</i>
Quality of the Optimisation	78
<i>Optimising</i>	<i>78</i>
Compiler Warnings	80
<i>Standard of the language</i>	<i>80</i>
<i>Security</i>	<i>81</i>
<i>Optimisation</i>	<i>81</i>
Path Settings and Linker Modes	82
<i>Generation of Programs</i>	<i>82</i>
<i>Library Path</i>	<i>84</i>
<i>Warnings</i>	<i>84</i>
<i>Optimiser</i>	<i>84</i>
Hunk Optimisations and Memory Settings	85
<i>Summarise Hunk</i>	<i>85</i>
<i>Manner of Memory</i>	<i>86</i>
<i>ROM Code</i>	<i>86</i>
Call of the executable program	87
<i>Execute Environment</i>	<i>87</i>
<i>Start From CLI</i>	<i>87</i>
<i>I/O Settings</i>	<i>88</i>
<i>Input/Output</i>	<i>89</i>
Save Project	89



OVERVIEW

Before we start with the project manager, I will tell you some basics of the system. One point will be a precise description of the start of StormC. You will certainly already have started the system, but the understanding of what happened and the function of the controls may still not be clear.

You have launched the development system, but the its behaviour and the settings to be made are still open and are subject to change.

Some words about the startup

The system is started by a program called StormCPP. This is a loader and its only duty is to start the other parts: StormC, StormED, StormLink, StormRun, StormASM and StormShell. As this could take a little while, a nice startup picture will be shown (if you are running an Amiga with less than 32 colours on the Workbench screen, there will only be a small window that shows you the progress.).

On an Amiga System with less than 32 colours on workbench screen, a little window opens to tell you the different programs being loaded.

Tooltypes

The attributes of a program will be stored in its icon. To change it please click on the icon "**StormCPP**" and select the menu item "**Icon/Information**". The following requester will open.



Default Setting

After the installation of StormScreenManager, a public screen named "StormC" will be defined, but it will not be used. You have to activate the corresponding attribute in the tooltypes of StormCPP. Please open the information window and delete the round brackets at the beginning and the end of the corresponding line to activate this attribute. At the next startup of StormC, this screen will be opened automatically. Now the Tooltypes (of the StormCPP icon) will be executed (if possible). The following options can be used:

QUIET

The **QUIET** tooltype will disable the picture show at start. The progress window which opens on Workbench screens with fewer colours (when not enough colours are available to show the pictures) will also be disabled.

HOTHELP

Upon activating the editor window you can press the <HELP> key to get context sensitive help from the AmigaGuide docs. If you would like to have the program "**HOTHELP**" doing this job you must enter the tooltype **HOTHELP**.

SAVEMEM

When memory is low, the use of this tooltype offers the possibility to only load parts of the system, to save memory. **SAVEMEM** prevents the loading of StormC, StormLink, StormASM and StormRUN. These parts are only loaded when they are needed. After being loaded these parts will stay in memory as long as there is enough memory.

GOLDED

The popular editor GoldED can be used instead of StormED. It is loaded from the drawer "**GOLDED**". As GoldED is used the same way as StormED, it is not the control office of the system, so it must be loaded memory resident.

Please configure GoldED at "**Config/Various...**" setting "**Various**" to "**resident**" and save the settings.

PUBSCREEN

With the **PUBSCREEN** tooltype, you can set the public screen name upon which all StormC windows will be opened. Like the Workbench, the public screen is shared among all programs and can be used as a target screen for another application.

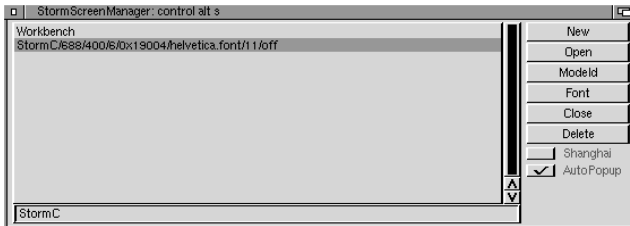
The creation of public screens is handled by a program that is independent of StormC and which is started separately. During installation process, if you have allowed the installation of **StormScreenManager**, a little program is copied in your **WBStartup** drawer to do this job.

The ScreenManager

The StormScreenManager is a little commodity which will be automatically started after each reboot. It will wait on background a public screen request and will react upon such a call



Of course, the StormScreenManager must know the names and the settings of the screens to open. Otherwise, the public screen requests will be ignored and StormC and every other program which asks for a public screen, will open their windows on the Workbench.



An entry for StormC is already defined in the StormScreenManager. To use this public screen, you just have to remove the parentheses on the same name entry in the StormCPP icon tooltypes.

To do this, click once on the StormCPP icon and select the menu item **"Information..."** on the **"Icon"** menu of the Workbench screen. Once you have the Information window, click on the PUBSCREEN tooltype, the string gadget will be filled with this tooltype. Then, you can remove the parentheses and press return to validate the changes. You can now start the development system by double clicking this icon.

Now a new screen is opened and all the StormC program windows will be shown there.

To modify the screen settings, you have to bring up the StormScreenManager window. Press <Control Alt s> and the window will popup as shown on the previous page.

Choose the StormC entry to modify its definitions. The name will be displayed in the string gadget and can be edited

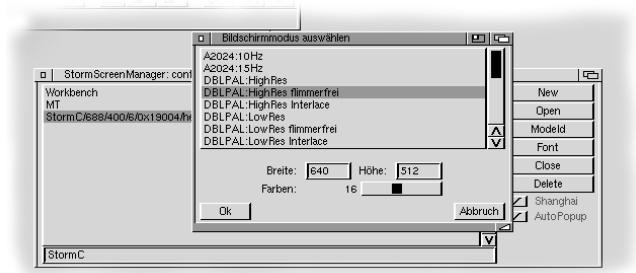


After the StormC installation process, you have to reboot your system in order to be able to use the StormScreenManager. To bring up its window, press the hotkey <Control>+<Alt>+<s>.



You can define your own hotkey definition to bring up the StormScreenManager window by modifying the appropriate tooltype. Please refer to the Commodities chapter of your Amiga Workbench user manual.

there. To change the screen mode, click on the **"ModeID"** gadget.



Choose the appropriate screen mode and depth for your monitor. Please, keep in mind that a minimum of 16 colours are needed for the StormC screen. When using 8 or fewer colours, the editor can not correctly display the text in the predefined colours.

On the right side of the StormScreenManager window, you'll find some other gadgets.

New

Creates a new standard entry in the public screen list. The entry will be appended at the bottom of the list, and you can modify the name in the string gadget and modify the other settings with the appropriate gadgets like ModelID and Font. You can also open and close it right away.

Open

Opens the selected screen in the public screen list. Screens already opened (ie: The Workbench screen) may not be opened again.

Font

Opens an ASL font requester for you to choose the font for your screen.

Close

Closes a screen opened by the StormScreenManager.

Delete

Closes a screen opened by the StormScreenManager and removes its entry in the public screens list.

In order to save all the changes applied to all the screens you have defined, you must select the menu item **"Save"** in the **"Project"** menu of StormScreenManager.



To finish with the screen manager a single click on the window close gadget will hide it. This will only close the program window and this is the equivalent of the **"Hide"** menu item of the **"Project"** menu. If you want to deactivate the program as well as remove it from memory you must select the **"Quit"** menu item in the **"Project"** menu.

Memory usage

Whenever starting the StormC development system, it always tries to load as many program parts as possible in memory to make them resident modules

If you are low on memory, StormC will still try to start up if there's enough memory to start StormShell and StormED

The programs StormC, StormLink and StormRun will be started via a Shell for any compilation and will be unloaded afterwards. This way, turnaround times will be higher.

Settings

Before opening any project or source, fundamental system settings should be created and saved.

Following settings can be made:

The initial size, position, display, and start address to show in the HexEditor window within the debugger can be set here.

HexEditor

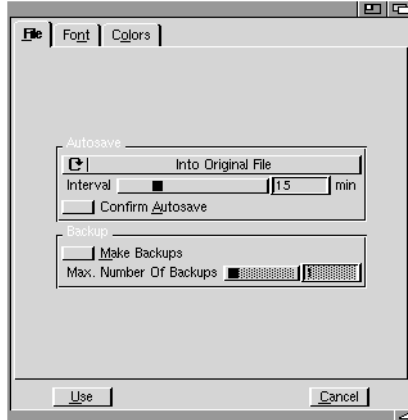
When the ProjectManager is in the "empty" state, you can set the font and paths for the source text editor. The other settings, such as syntax colourisation and entries linked with source files, will be saved in the predefined icon. More information about this later.

Source text editor



Menu item names followed by 3 points (i.e.: "...") indicate that this is not a direct function, but that more information is needed such as a confirmation or special settings. A good example is the menu items "**Save**" and "**Save As...**". Selecting "**Save**" will save the file immediately. "**Save As...**" will let you change the name before performing the "**Save**" function.

Selecting the menu item "**Settings/Editor...**", the following requester will pops up.



This window can be sub-divided in order to separately define the File, Font and Colour settings.

The upper paging gadget lets you swap between the File, Font and Colour settings.

The picture above shows the file settings editor. This is where you can define the AutoSave settings and if you want to keep backups of your source files.

Automatic storage

With the cycle gadget within the "**AutoSave**" group, you can swap between the following possibilities.

Never

There will not be any automatic storage.

Into original file

At this position the storage will be done overwriting the original file. The flag indicating file changes will be removed once the storage procedure has completed successfully.

Into extra file

If the automatic storage is to be done in an extra file, a second file will be created with the suffix "**;0**". The AutoSave will still be performed in the same file.

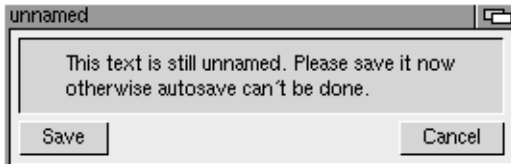
The "**M**" flag will not be changed. It means that the secure file will be more recent than the original file after a possible crash. After a restart, if you load your original source, a



requester will pop up to tell you that the AutoSave file is more recent than your original file.

Automatic storage of unnamed files

If a file is still unnamed at the automatic storage timeout, a requester will pop up to give you the chance to enter a name.



An ASL filerequester will let you enter a name for this file. If you cancel the ASL requester, it will pop up again after the AutoSave timeout.

Automatic storage will only be performed if the text has really been modified. Of course, whether any changes have been made can be seen by the "M" flag in the status bar of every window.

If you have checked the "**Confirm AutoSave**" gadget, a requester will ask confirmation before saving the file.

Automatic backup copy

If you have checked the "**Make backups**" gadget in the "**Backup**" group, as many copies as selected will be performed. The files will have the suffix ";1" to ";9" depending on the selected copies number.

For instance, setting the number of backup copies to 2, the first time a file is saved (manually or automatically) will rename the original. It will have the ";1" suffix and a new original file will be created. The next time that file is saved, the file with suffix ";1" will be renamed with the suffix ";2". The original file, as before, will be renamed with suffix ";1". A new original file will be created. The third time, it will react exactly as described before.

Selecting text font

In the general editor settings window, choose the "**Font**" position for the paging gadget.

The font selected here will be used for all the editor windows. With the cycle gadget in the "**Text font**" group, you can choose between the screen font and a custom font.

The default settings of StormC will use the screen font. This is the font you selected with the font preferences of the Workbench for "**Default text font**".

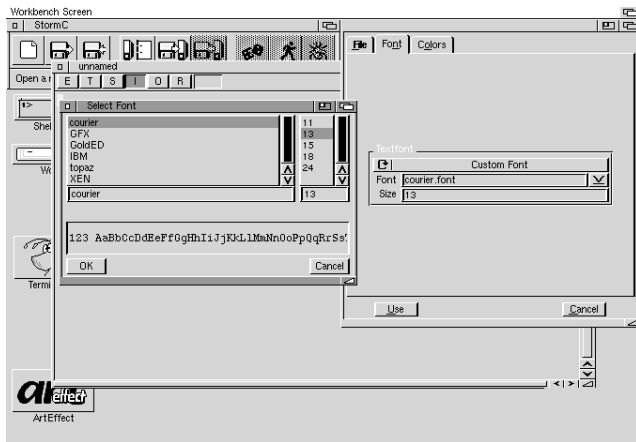


The editor window can only display properly

with monospaced fonts. This means that only non-proportional fonts (where the width of all the characters is the same) may be used.

Clicking on the cycle gadget until "**Custom font**" is shown, will let you select the font and its size.

Clicking on the pop up gadget to the right of the string gadget will pop up an ASL font requester to choose the font and its size. This requester will only display the non-proportional fonts.



A detailed description of the use of the ASL font requester can be found in your AmigaDOS user manual.



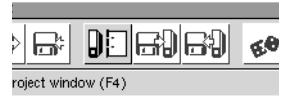
ORGANISATION OF A PROJECT

Each program you create with StormC should be defined as a project. All projects contain at least the source, all pertinent header files, the name of the program to be created and the options for the compiler, linker and debugger.

Moreover you can add ARexx and Installer scripts, documentation as ASCII or AmigaGuide files, and other program related files to your project.

Creation of a new project

To create a new (empty) project you have to click on the icon "**New Project**" in the Toolbar. When the Toolbar is activated you can select the menu item "**New**" from "**Project**" as well.



The default settings for a new project will be loaded from the file "**template.¶**" which can be found in "**PROGDIR:**" which is the drawer StormC was started from. If you have not changed the installation (which you should not do), it is the drawer "**StormC:StormSYS**".

The template of the project contains predefined options for the compiler, linker, debugger, and the standard libraries.

Of course you have the option of changing all default settings, so when creating a new project your preferred settings will be used. More information on the definition of project templates can be found in the "*Own Project - Template*" section of this chapter.

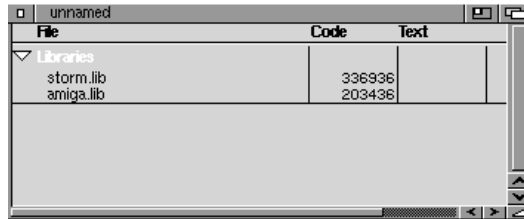


If there is no file "template.¶" in "PROGDIR:"

StormC uses its default settings for the new project. In this case the project will not contain any settings for standard libraries and an error message will not be created.

Setup of projects

When creating a new project, a new project window appears on the screen in which the standard libraries can be found.



Horizontally the project divides into three sections:

File

The horizontal section "**file**" shows, as the first entry, the title of the vertical section. In our example it is the section "**Libraries**". In front of the title you can see a small triangle which points to the bottom. The triangle symbolises that this section is not folded so the following entries up to the separating vertical line are part of the "**Libraries**" section. In our example the names of the libraries are "**Storm.Lib**" and "**Amiga.Lib**".

Code

On the right beside the file section there is a code size section. Here you will find the size of the output code of compiled or assembled sources, linked programs and libraries. This section remains empty for header or other files which are not translated by the compiler.

Text

On the right side of the window you will find the text size of script files (e.g. ARexx and Installer scripts) or other ASCII files. In our example this section is empty because these are libraries without source, so their size is unknown.

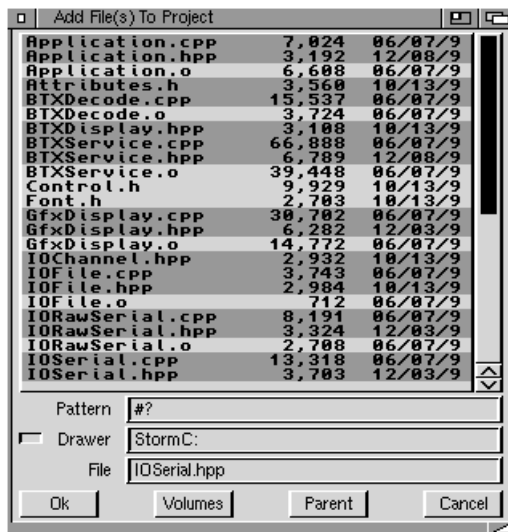
Project Sections

If there were no project sections all data of a project would be disorderly. You would only recognise the meaning of a file by its extension, if it had one. To get a better idea of which files are which, we included the project sections.

Adding Sources

Sources are all ASCII files of the project. A fine distinction will be made by the file extension.

When choosing the menu item **"Project/Add File(s)"**, you will have the option of selecting one or more files to add to your project. The ASL filerequester is used for this.



A description of the usage of the ASL filerequester

can be found in your Amiga manual.

When choosing several items at a time, they will be inserted alphabetically. If you would like to have them in another order you must select them individually.

Add Window

The menu item **"Project/Add Window"** is available only if there is an active editor window. Choosing this item will add the filename of the text of the activated editor window to the project.

If you didn't give a name to your text file before this, you will get the opportunity to do it now.

Adding Libraries

To make adding of libraries easier, there is a special menu item called **"Project/Add Librarie(s)"**. The ASL filerequester will come up with the library path you selected in project settings so you can add libraries very quickly.



It is important to know that **"Amiga.Lib"** contains functions with the same names as functions in the standard ANSI library, but there are differences in the parameters. When linking **"Amgia.Lib"** before **"Storm.Lib,"** these functions will be taken from **"Amiga.Lib,"** which could cause your program to crash, or otherwise act incorrectly.

Choosing a program name

The last item of the group is for selection of the program name. If you didn't choose a name for your program the linker will automatically create a file called **"a.out"** in the project path when you link your code.

After selecting the menu item **"Project/Select Executable..."** the ASL filerequester will show up. Now you can choose a drawer and a name for your program.

Please note that there will be a file with the extension **".link"** in this drawer when you are using the debugger. This file contains data about the modules of the program and their paths.

Drag&Drop

Of course you may drag files from the Workbench to the project window. If you drop an icon into the project window it will be added to an existing section or a new one will be created.

Controls of the Project Window

You certainly have recognised the folding of the project section. The small triangle indicates if the following section is folded or unfolded (so you can see all its entries).

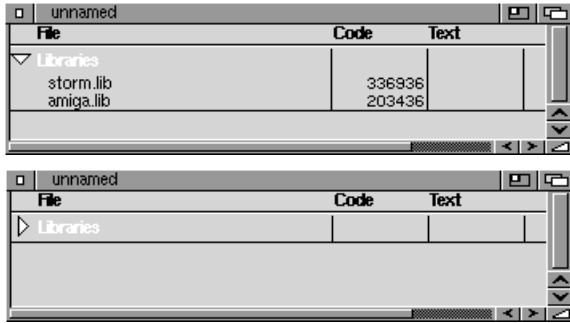


Instead of using the normal linker libraries you can use the MaxonC++ compatible manner of using a logfile for linking. Please note that the logfile will be used after the normal linking. If you are using normal libraries as well, they will be linked first and then the logfile will be executed.

Please note that if you are using multi-selection this list will be inserted alphabetically. That means that "Amiga.Lib" will be linked before "Storm.Lib" which could cause trouble

Folding of Sections

If a section is not folded the triangle shows to the bottom and all entries are visible. The following pictures will show the unfolded and the folded libraries section.



To fold/unfold a section you must click on the triangle next to the section name. The same actions will take effect if you double-click the section title.

But there is more functionality in the project manager than to fold/unfold sections.

Showing Sources

If you double-click a source in the source section it will be opened in the editor. During loading there will be a fuel gauge that shows you the progress.

But not only sources in the Source section can be loaded that way. You can load every entry in the following sections into the editor by a simple double-click:

- Header files*
- ASM sources*
- ASM header files*
- Documentation*
- ARexx scripts*
- AmigaGuide files*
- FD files*
- Locale catalogues*

Starting a Program

By double-clicking an item in the section "**Program**," you will start that program.

If you hold the <ALT> key while double-clicking such an entry, you will start the program in debug mode.

Open a Child Project

Of course there can be projects within projects, so it is easy to administrate the source for shared library development, or multiple program projects, as well. With a simple double-click on such an item in the project manager you can open the project.

Keyboard control

You have certainly recognised that the section title and the items will stay selected (inverted) after selection. This is an indicator for keyboard control.

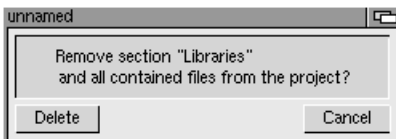
Cursor keys

With the keys <UP> and <DOWN> you can move the inverted beam up or down. If there is a horizontal slider at the bottom of the project manager window you can move it with the keys <LEFT> and <RIGHT>.

Return

The <Return> key has the same action as a double-click with your mouse. If the inverted beam is at a section title and you hit <Return>, the section will be folded/unfolded. When the inverted beam is over a document, a source file, or anything that could be edited in an editor, it will be opened there if you hit <Return>.

Deleting Items of a Project



If you want to delete a whole section of your project you must select the title of it. A requester will appear, to verify whether you really want to delete this section or not.

Organising files of a project

The biggest advantage working with a project manager is the order of all the parts of your project. This will normally be

 If you are deleting a file the way described here this will not really delete the file. It will only delete its appearance in the project manager. The file will still be available on your hard drive. Of course you can delete files or complete sections in the project manager. You might select the menu item "Edit/Delete" or you can press or <Backspace>.

done automatically, but I will show you some methods to do it even better.

As you know it is very easy to create a new project. You only have to click on the icon **"New Project"** on the toolbar and a standard project template will be opened. This project has all settings that are needed to start, but one thing is missing: there is no file name and you have not defined the place to save it.

Every project should have its own drawer. Naturally you can store more than one project in a drawer, but you shouldn't do this, as then you will not know which header files and documents belong to each specific project.



A simple click on the "Save Project" icon on the toolbar is enough to save your project. As an alternative you can select "Save" or "Save As..." from "Project" as well.

The first step after creating a new project is to save it. Even if you did not add anything to it this is the best way to give your project a name and its own drawer. You can do it with one step. Hit the **"Save Project"** icon on the toolbar. An ASL filerequester will appear. Now type in the name of the new drawer and the name of the project e.g. **"New Drawer/My Project"**. Now you have created a drawer called **"New Drawer"** and a project called **"New Project. ¶"**. The extension **".¶"** will be added automatically.



Please note that you have to enter <Return> first. Unfortunately a click on the "Save" icon will not test whether you want to create a new drawer.

As soon as you hit <Return> you will be asked if you want to create a drawer named **"New Drawer"**. Confirm it with **"OK"** and press <Return> or click **"Save"**.

Project specific headers

Class and structure definitions that are used in several modules should be collected into a common header file. In our example debugger tutorial, there is a project specific header file called **"address.h"**. When you look at the source to **"main.c"** and **"address.c"**, you will see that these header files are called with quotation marks and the standard headers with angle brackets **"<>"**.



The options of the standard include drawer will be described in the following sections.

Looking at the quotation marks on **"#include,"** the compiler decides whether to load the file out of the standard include drawer or from the same drawer as the source.

The following line will cause **"stdio.h"** to be loaded from **"StormC:Include"** by the compiler. If it does not find it at this location, all alternatives will be tested before an error report appears.



```
#include <stdio.h>
```

This line

```
#include "address.h"
```

causes the compiler to look in the same drawer as the source for "address.h."

As a result, you should not use a special drawer for project specific header files. It is better to store them in the same drawer as the source. You should not store them in the standard include drawer, either. First of all, this does not help you keep your files in good order, and it also prevents the project manager from knowing about the dependence between source and header files.

Header files which are in angle brackets "< >" will cause no automatic re-compilation if they are changed in the editor. However, the dependency of header files which are stored in the source directory, and which are enclosed with quotation marks, will be regarded. That means that after a change of these header files, the sources in which the headers are included will be recompiled automatically.

Documentation, scripts ...

You should use separate drawers for documentation, AmigaGuide files, Installer and ARexx scripts if there are more then two files per section.

You don't have to keep this order, because the project manager will show everything very clearly (divided in sections). But from time to time you will look at your old projects, and then you will see the advantage of good structure. The more you work on the structure of your project the easier it will be to find the files in old projects.

Makescripts

The rules behind a "Make" are essentially very simple. First of all, all files in the project are checked to see if they need to be recompiled.

In the case of a C source file this means that the file dates of its object and debug files are compared to that of the source text and of any header files that it may **#include**. If any of

these is newer than either the object or debug file, the source file needs to be recompiled.

The source file also needs to be recompiled if one of the header files has been changed by some other action by the compiler. This may be the case for instance when the "**catcomp**" program is used to generate a header file from a Locale file.

Once it has been determined which files are to be recompiled or re-linked, each of them is handled by sending the corresponding ARexx commands to the StormC compiler and the StormLink linker. These commands are then executed in turn.

Makescripts are used when other files than just C and assembler sources need to be translated:

The "**Select translation script...**" menu option lets you enter an ARexx script for the active project or - if a section title has been selected - for all files in a section. These scripts make it possible to invoke external compilers such as eg. "**catcomp**" to compile Locale files automatically.

They are called by the project manager whenever the project is to be recompiled. Makescripts should have filenames ending in ".**srx**". Files with this extension to their names are also included in the ARexx section.

Selecting the "**Remove translation script**" menu option will remove the makescript from a project entry or from all entries in a project section.

The rules for determining whether a project file that has a makescript attached should be recompiled, are essentially the same as they are for C source files.

A file is always recompiled during the first "**Make**" after a makescript has been added to it.

As an example of what a makescript looks like, the "**catcomp.srx**" script is explained below:

The script's arguments are the file name (ie. the path to the project entry) and the base project path. Both are enclosed in quotes to allow the use of spaces.



The argument list must be terminated by a full stop, so that any additional arguments that may be passed by future versions of the compiler will be skipped.

```
PARSE ARG ''' filename ''' ''' projectname ''' .
```

The object filename is constructed from the filename argument. This isn't necessarily a file that is going to be linked and whose filename ends in ".o", but simply the file that is to be created. Catcomp happens to create a header file.

```
objectname =
LEFT(filename, LASTPOS('.cd', filename)-1) || ".h"
All output is sent to a console window.
```

```
SAY ""
SAY "Catcomp Script c1996 HAAGE & PARTNER GmbH"
SAY "Compile " || filename || " to header
" || objectname || "."
```

In order to allow the Project Manager to determine when the file should be recompiled, the object filename must be coupled to the project entry. If this statement were to be omitted, the makescript would be called for every **"Make"**.

A maximum of two object filenames may be given as follows:

```
OBJECTS filename objectname1 objectname2
```

These names are then attached to the entry and the files are checked when recompiling.

The **OBJECTS** statement should not be used if the makescript is used for calling an assembler in the section **"Asm Sources"**. For this section the object names are derived automatically.

*See also the script
"StormC:rex/phxass.srx".*

```
OBJECTS filename objectname
```

This is where the translating program is called. Error messages are printed in the console window.

```
ADDRESS COMMAND "catcomp " || filename || " CFILE
" || objectname
```

As **"catcomp"** creates a header file, it is advisable to enter this header file into the project. The **QUIET** parameter

represses any error messages in case the header file should already be included in the project.

ADDFILE objectname QUIET

/* End of makescript */

Almost any makescript can be built along these lines. Another statement may be useful in some cases:

DEPENDENCIES filename file1 file2 file3 ...

This statement connects the project entry to further files whose dates will be checked to see whether or not the makescript should be called. The file named in the project entry itself will always be checked and need not be specified using this statement. Using this statement makes sense in cases where the script involves any extraneous files (the StormC compiler for instance uses it to declare any header files that a source file includes with **#include "abc.h"**; note that this is not done for headers included with **#include <abc.h>**).

Makescript settings are ignored for the project section that contains C sources; these files are always run through the StormC compiler. The section containing assembler source files on the other hand allows the use of makescripts - although it will use the built-in default rule for StormASM (which in turn invokes the PhxAss assembler) if no makescript is set.

Passing arguments to makescripts

The script receives the filename (that is, the path to the project entry) and the project path as arguments. Both paths are enclosed in quotes to allow the use of whitespace in file or directory names.

Next comes a numeric argument whose value indicates whether the object files should all be written into a single directory.

0 means that the object file should be stored in the same directory as the source file;

1 means that the object file is to be stored in the object-file directory.



The name of the object-file directory - quoted like the other paths - is passed as the next argument (regardless of the value of the previous argument, ie. even when the preceding numeric argument is 0).

The object-file directory is only interesting to programs that generate code. Source-generating makescripts (eg. "**catcomp.srx**") will always write their object files to the same directory that the file in the project entry resides in. Thus only assemblers and other compilers really need to care about the object-file directory.

Makescripts for assembly source files are an exception in that they take an additional third argument: The name of the object file. This name must be used when creating the assembler object file. The path to the object-file directory is already included in this name, if necessary.

The argument list must be terminated by a full stop so that any additional arguments that may be passed by future versions of the compiler will be skipped.

A complete PARSE statement for makescripts (other than one for assembler sources, as explained above) is composed as follows:

```
PARSE ARG ''' filename ''' ''' projectname '''
useobjectdir ''' objectdir ''' .
```

For an assembler makescript this would be:

```
PARSE ARG ''' filename ''' ''' projectname ''' '''
objectname ''' useobjectdir ''' objectdir ''' .
Ready-made makescripts
```

The directory "**StormC:Rexx**" contains several ready-to-use makescripts. You may want to adapt them to different uses and situations:

Assembler scripts

Makescripts for assemblers differ from other makescripts in that they may not contain the **OBJECTS** statement.

"phxass.srx"

This script translates an assembler file using the PhxAss assembler. This script is really superfluous because the assembler is supported by the StormShell directly, but may be useful if you want to use different assembler options.

"oma.srx"

This script translates an assembler source file using the OMA assembler.

"masm.srx"

This script translates an assembler source file using the MASM assembler.

Other scripts:

"catcomp.srx"

This script translates a Locale catalogue file by invoking the program catcomp.

"librarian.srx"

The StormLibrarian can also be controlled through makescripts. A project entry in the **"Librarian"** section can be loaded directly into StormLibrarian by double-clicking it with the mouse, or the linker library can be created simply by double-clicking it while keeping the Alt key pressed. But if a project should always create a link library, the use of makescripts is recommended. The list of object files is created in StormLibrarian as usual. The makescript then invokes the StormLibrarian, which not only automatically generates the library, but also declares the linker library as an object (using **OBJECTS**) and all object files in the list as dependant files (using **DEPENDENCIES**). After the first Make this will cause the linker library to be created anew whenever any of its C or assembler source files has been recompiled.

The library will also be recreated if its list of object files has been modified using the StormLibrarian.

"fd2pragma.srx"

This makescript translates an FD file into a header file containing the necessary **"#pragma amicall"** directives for a shared library. This script shouldn't normally be necessary



as the linker writes this header file automatically whenever a shared library is linked.

Saving of the paths

When saving a project you will give a filename and a path for storage. Before you save a project all files are stored with their full path which is sometimes very long. After the save, all these paths are shortened to relative paths. This means that all files that are stored in the same drawer will be stored with their name only. The path is no longer relevant.

PROJECT SETTINGS

Every project has its own settings for the compiler, linker and RunShell. Maybe that doesn't sound difficult, but there are many settings and connections that should be thought about before compiling a new project.

First there are some principle settings, e.g. default paths for Includes and linker libraries, standard definitions and the settings for the program start.

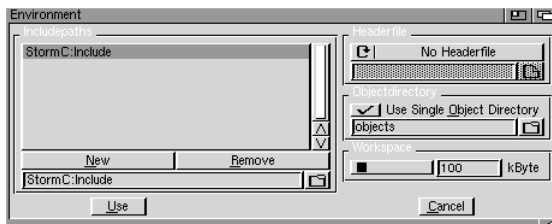
Next, the more complicated settings for code generation. The global settings for a project, whether the compiler will run in C++ or ANSI-C mode, can be changed for every source file.

Many of these settings will be set to a default if you create a new project. Some of the settings are project specific and must be set according to the needs of the project. As an example: before starting the linker you must decide whether to build a driver or a shared-library. If you want to create a shared-library, but the settings are for a normal program, the result will not be satisfying, but you can correct this very easily by changing the settings and starting the compiler again.

The following sections will explain the individual controls of a project. You should open a new project and choose the menu item **"Settings/Project Environment..."**. The following requester has a cycle gadget to change between eight different pages. The gadgets **"OK"** and **"Cancel"** will accept or cancel all these settings. You should use **"OK"** to accept all these settings or click **"Cancel"** if you don't want them to be accepted or if you just wanted to have a look at them.

Paths and global settings

The first control is the settings for the includes.





Include path

The pre-processor reads the definitions for the standard functions out of the include files. You use the pre-processor directive **"#include"** to include these files into your source. As you may know, there are two possibilities for this: You can put the include file name in quotation marks like **#include "this_one"** and the pre-processor will look for it in the same drawer as the source, or you can use the angle brackets like **#include <this_one>** and the pre-processor will recognise them as standard include files, and look for them in the predefined directory. The predefined directories are listed in the listview you will see at the left side of the requester. You can define more than one directory for searching standard include files.

With a simple click on **"New"** you can create a new entry in the listview. You can type the complete path into the string gadget under the **"New"** button. If you do not know the exact path, or simply don't wish to type it, you can use the ASL filerequester to look for it by clicking the icon next to the string gadget.

New

Click on **"Remove"** to delete the selected entry from the list.

Remove

Header Files

One way to increase the speed of the compiler is to create and use pre-compiled header files. The pre-compiled header files are always present after you create them once. Another advantage of the pre-compiled header files is that all files are assembled to one big file which can be loaded faster.

To tell the compiler where the header files are end and your program starts, you must mark it with a **"#pragma header"** statement. One simple way is to put the **"#includes"** of all headers you didn't write yourself (e.g. OS includes), or which will not be changed, at the beginning of your source. The **"#pragma header"** statement should follow. After that, list your own header files and the rest of your program.

The pragma statement has no effect until you activate **"Write Header File"** in the cycle gadget **"Header Files"** and start the compilation of any changed modules. As soon as the compiler reaches the pragma statements it will stop compiling your source, and the pre-compiled header file will be written with the given name to the predefined drawer.

Before you start the compiler again you should switch the cycle gadget "**Header Files**" to "**Read Header File**". Next you should enter the path to the pre-defined headers into the string gadget below. You can use the ASL filerequester for that by clicking on the icon next to the string gadget. The compiler now uses the pre-compiled header file and looks for the statement "**#pragma header**" to start further translation.

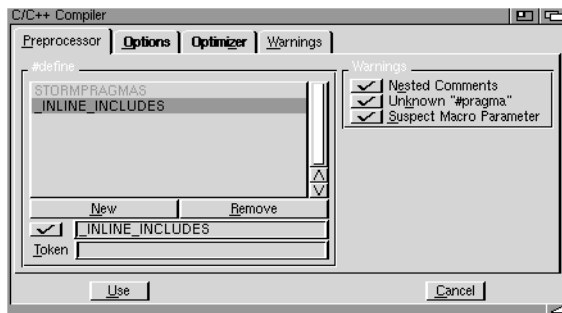
Working memory - Workspace

The compiler will need some free RAM for code generation and assembling, but StormC does not know how much in advance, so you must choose a value for workspace. If the workspace is not large enough, the compiler will stop and give you the error message "**Workspace Overflow**".

A value of 100 KB will be enough for most of your programs. But if the compiler gives you an error message you should raise this value and start compilation again.

Definitions and Warnings

With these controls you can choose pre-processor warnings, and you can select the predefined pre-processor symbols.



#define

In this list you can predefine pre-processor symbols. Every item in this list will be treated the same as if you had entered a "**#define**" line at the beginning of your source.

An example: You have inserted several `assert()` calls in your program and you included the file "**<assert.h>**". To disable these calls, you must add "**#define NDEBUG**" somewhere at the beginning of your source, before the "**#include**



`<assert.h>` line. Or, you may simply insert `"NDEBUG"` into the list of predefined pre-processor symbols and mark the gadget in front of the string gadget.

To disable a special pre-processor symbol for the next compiler run you can simply mark it with the mouse. It appears in the string gadget below the listview and you can enable/disable it by clicking the gadget on the left side.

Enable/Disable

The controls for `"#define"` are nearly the same as those for the `"#include"` controls in the previous section.

This will create a new entry in the list and the string gadget will be activated. Type in the name of the new definition e.g. `"STORMPRAGMAS"` and press `<Return>`. The new pre-processor symbol will be inserted in the list now and it is activated. If you interrogate `"#ifdef STORMPRAGMAS"` you will get the value `TRUE`.

New

With **"Remove"** you can delete the marked item from the list.

Remove

You can not only define a symbol; you can also assign a value to it. You can enter any alphanumeric character into the string gadget. Unfortunately, the compiler only accepts one character per symbol, so the entry `"(11-5)"` will only take the `"("` and ignore the rest. More complex definitions will need to be done directly in the source, but you can use it for symbol definitions for flags like `"NDEBUG"`.

Character

Even at runtime of the pre-processor, there can be language constructs that are syntactically and semantically correct, but that might be programming errors.

Warnings

This warning appears when embedded comments, e.g. `"/** This is a /* comment */"`, are found, since there might be a missing `"*/"` at the end of the comment.

Nested Comments

Pragma statements are independent of the compiler. But if your sources were made for other compilers they might create confusion. Normally the compiler ignores every pragma it does not know, but a warning will give you the chance to see whether you made a typing mistake.

Unknown #pragma

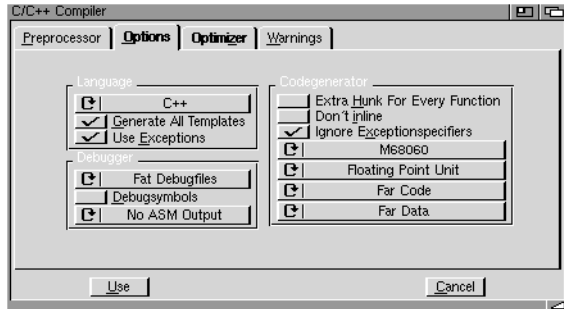
If a macro argument extends over 8 lines or 200 expressions, the compiler assumes that you have forgotten a closing

Suspicious Macro Parameter

bracket. So, there will be a warning, and you will not have to look for this error endlessly.

ANSI-C/C++ Settings

With these settings you can control code generation and debug output.



Source

You can choose between "ANSI-C" and "C++" mode. If you are in C++ mode you can make some additional C++ settings.

Template Functions

Create All Templates

Templates are a very useful and easily understandable construct of the C++ language, but they are not easy to use. Essentially function templates have some surprising properties. Normally you can not say if the definition of a template is correct or not. Maybe you can think of a discretionary function template for a sorting algorithm (e.g. quicksort). The type of the element of the vector that has to be sorted is normally a template argument.

The programmer can solve the problem by implementing the needed function for the datatypes by himself. Since you are working with more than one module on a bigger project the compiler can not know which of the not-created, but needed functions exist as finished ones anywhere within the project. The compiler has two different strategies to solve this problem. The first one will be activated by choosing the entry "**Create All Templates**". The compiler assumes that you didn't write your own templates, so it generates all templates which are needed, but are not already defined. If an error occurs during this process it will be reported. Functions that are created but that are not needed will be deleted by the linker, later.



Another strategy will be selected if you do not mark "**Create All Templates**". In this case all templates will be created too. If there an error is found in one of the functions the process will be halted. There will be no code generation for the faulty function, and the programmer will not be bothered with the error message. The compiler assumes that all templates that can not be created without an error are defined elsewhere in the project.

I advise you to switch on this option during program development, because only in this case will you will get a detailed error message about errors in the template. But if you are working with finished template libraries or if you are porting an existing program, there might be a need to switch off the generation of templates.

If you ask asking yourself why you have to care about the right strategy, I will tell you how to implement a function template correctly:

- Do not create templates the first time you compile your code.
- The linker tries to link the project and generates a list of all undefined functions.
- In the final compiler run, all modules that might contain a helpful template definition will be compiled anew. With the help of the list, generated by the linker, it is clear which functions must be generated.

Such an implementation would not only be very large-scale, but also very slow because some modules must be compiled twice every time.

Exceptions

Sometimes it is an advantage if you compile a "classic" C++ program that does not use exception handling in a special compiler mode. The related keywords ("**catch**", "**throw**", "**try**") will not be recognised, and in the created code there is no book keeping of the needed destructor calls. If "**Use Exceptions**" is switched off, exception handling is not possible any more.

Using Exceptions

Debugger

If you choose to create a debug file the compiler will generate a file for each translation unit from which the debugger could take the needed information. There are two possibilities: "**Small Debug Files**" and "**Fat Debug Files**".

Small Debug Files

Small debug files contains only a few pieces of data about variables. They are stored in the debug files of every module. There is no information about "**#include**" files which are in angle brackets within a debug file.

Fat Debug Files

These files contain all data types, structs, and class members. The debug files of every module are accordingly big.

Create Symbol Hunks

Instead of the StormC source-level debugger you can use a symbolic debugger like "MetaScope" as well. To get some important information you should create the so-called "**symbol hunks**".

ASM source without C source

If you selected this option the compiler will generate an assembler listing for every file it has compiled successfully. The assembler source will be stored in a file with the same name as the C source but the ending ".s".

If you are familiar with other compilers you might wonder about the "obviously" well optimised assembler code. To clarify this: The compiler generates intermediate code in an internal binary format that looks like assembler source in some way, but that has nothing to do with an ASCII file. The "assembler" will translate this intermediate code to real machine code while doing some optimisations.

The assembler source generated by StormC correspond exactly to the code which is translated to machine code directly. So you do not need the help of a re-assembler to look at the quality of the generated code.

ASM Source With C Source

If you want to have the corresponding C source (as a comment) within the assembler source you should select this option.

Code Generation

Extra Hunk For Every Function

A hunk will be generated for every global variable and every function during code generation. As a result the linker can eliminate every unused function and the code size will



decrease. This is very practical if you want to program a library, but it has a price: functions of a translation unit can not use identical strings collectively and they cannot call each other via PC relative addressing. This can result in a larger, slower program.

Processor Specific Code Generation

The proven MC68000 processors are a dying species in the Amiga world. 32-bit processors (68020 to 68060) and Floating Point Units (68881 and 68882) have prevailed. So programmers have new and powerful machine commands which give them a sometimes astounding increase in speed. The 32-bit processors have some important features like the operators for longword multiplication and division and for the handling of arrays and bit fields.

By using the cycle gadget you can tell the compiler which code it should generate (68020/30/40/60).

Programs which do many floating point operations will be sped up by using an FPU (Fast Floating Point Unit). The option **"Use FPU"** will generate special code for the FPU.

If you switch to **"68040/60"** optimisation the use of the FPU will be activated automatically. This is because both processors have a built-in FPU.

StormLink supports "near code". In this model, pc-relative addressing of functions is used to get faster code and smaller programs. If the NearCode hunk gets bigger than 32 Kbytes, StormLink inserts a **"jump chain"** between the single hunks to make a call with more than 32 Kbyte address range possible.

A program which is created in the small data model has a single hunk which contains the pre-initialised data of the program and the uninitialised data (BSS). Access to this hunk is handled by address register relative addressing. These accesses have the advantage of being faster and shorter than 32-bit absolute accesses.

In the small data or code model, only SHORT will be used for an address. These are 16 bit or 2 bytes. This makes the Program shorter and faster but the program can only be 64 Kbytes in size. The short addressing is possible though a trick. One register of the CPU, normally A4, points at a place



If you generate code for 68020 and above or for an FPU these programs might not run on your old Amiga any more. They will certainly crash suddenly.

Small and huge code model

Small and huge data model

in the data area of the program and all addresses can be called by an offset to this register.

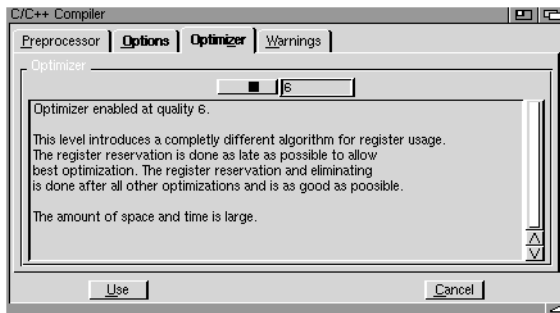
Small data model (a6)

"**Small Data Model (a6)**" is a format that is available in StormC only. It is mainly the same as the NEAR format, but the base register is A6 instead of A4.

In a shared library the program has no elegant method to get the value of the base register, but it does get the library base in register A6.

Quality of the Optimisation

To improve the quality of the code requires a lot of RAM and an increase in the translation time.



Optimising

At this time the compiler knows 6 levels of optimisation. You may select the level with the slider or by entering the number directly.

The following optimisations will be done by the respective levels:

Level 1

The first level optimises basic blocks. A basic block is a sequence of code statements which do not contain any jumps. Successive blocks will be joined to make later optimisation easier. Unused blocks (which are not jumped to) will be deleted. This step will be repeated until the code can no longer be improved. The joining of successive blocks and the deletion of blocks never used will also occur in higher levels.

Level 2

Useless statements, such as assigned variables that are never used, will be detected and deleted.



Automatic registerisation of temporaries and variables will be used if possible. **Level 3**

Assignments to variables that are never used will be deleted, and the whole program will be checked again until no redundant assignments can be found. **Level 4**

An example:

At this useless function

```
void f( int i)
{
  int j = i+1;
  int k = 2*j;
}
```

the second assignment will be recognised as useless. From level 4 on the code will be checked again so the first one will be deleted too.

During M680x0 code generation, redundant MOVE commands will be eliminated. **Level 5**

So

```
move.l 8(a0),d2
add.l  d2,_xyz
```

will become

```
add.l  8(a0),_xyz
```

During expression evaluation, temporary variables will be created at code generation for provisional results of any kind. They may be put into processor registers later. Temporary variables from lower levels will be recycled if possible, to keep the number under a certain boundary. **Level 6**

The expression "**a*b+c**" will become intermediate statements like:

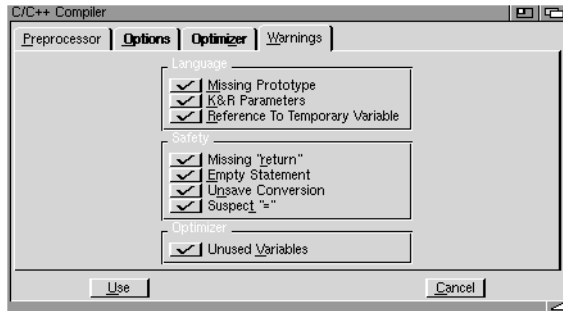
```
h1 = a*b
h1 = h1+c
```

From level 6 on these temporary variables will not be re-used principally. So there will be code like

```
h1 = a*b
h2 = h1+c
```

In a later optimisation step, a test is made to see if it makes sense to put "h1" and "h2" into the same register. Because an increasing number of temporary variables can cost a lot of time and RAM, this can be an expensive optimisation, but it helps make the best use of the CPU registers.

Compiler Warnings



Some constructs and expressions are syntactically and semantically correct, but they are a little strange and might be the consequence of a programming mistake, so there are eight warnings which can be switched on and off as personal preference.

These are the warnings:

Standard of the language

Missing Prototypes

If a prototype is not declared for a function, there will be a warning in C mode. In C++, this is an error.

Old K&R Parameter

"K&R" means "Kernighan & Ritchie" the Pre-ANSI pseudo-Standard. With this option you will get a warning if a function is declared in this old style.

Reference to temporary Variables

According to the C++-2.0-Standard, it is an error if a Reference will be initialised to a non-constant type with a non-L-value and a temporary object must be inserted. As this rule did not exist in the first standard there is no real error message but a switchable warning.



An example:

```
void dup(int &ir)
{ ir *= 2; }

void main()
{ int i; dup(i); // OK
  long l; dup(l); // ATTENTION! }
```

As **"I"** must be converted from **"int"** to **"long"** a temporary object must be inserted. The programmer will be surprised because at the second **"dup"** call there is a side effect on the argument. So the new C++ rule is very sensible.

Security

If there is no **"return"** statement in a function which has non-**"void"** return type, then there is certainly something wrong. You should always switch this warning on.

Missing "Return"

A statement like **"69;"** makes no sense, so there is probably something wrong with it. A well-known error is the call to a parameter-less function without an argument list, e.g. **"test"** instead of **"test();"**.

Empty Statement

This option is warning if there is an implicit type change (without CAST) from integer to floating point types.

Unsure Statement

This is a popular construct **"if (a=b)..."**. It is correct and many programmers like it very much, but it is a beginner error as well if it is used instead of **"if (a==b)"**. If this option is active there will be a warning if an **"="** follows a logical expression e.g. **"if"**, **"while"** or **"do"** and operands like **"|"**, **"&&"** or **"!"**.

Suspicious "="

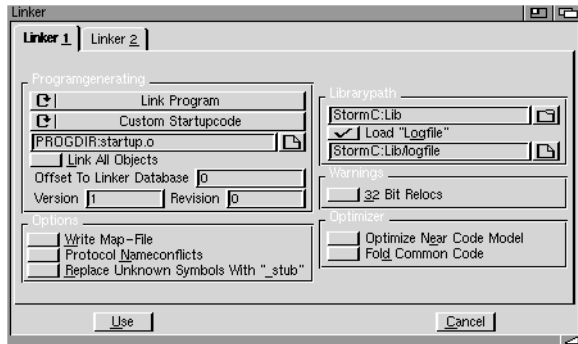
Optimisation

There will be a warning if there is a declared variable that is never used, or if a variable is used but not initialised.

Unused Variables

Path Settings and Linker Modes

The Settings of "**Linker Options 1**" set the defaults of how a program has to be linked.



Generation of Programs

Linking Programs

With the first cycle gadget you will choose the manner of program of the project. "**Link Program**" will use the standard library. If the next cycle gadget says "**StormC Startup-Code**" the file "**PROGDIR:startup.o**" will be linked to the program.

No Linking

The Linker will not do anything. After the compiler run, the project will be "ready compiled". The object and debug files will be stored, but no executable will be created.

Link as Shared Library

The linker creates a function table out of the indicated FD files and imports the standard shared library functions "**LibOpen()**", "**LibClose()**", "**LibExpunge()**" and "**LibNull()**". If "**StormC Startup-Code**" is selected the file "**PROGDIR:library_startup.o**" will be linked to the program.

StormC Startup-Code

As said before, the standard Startup code will be linked to the program when this option is selected. The Startup-code can be found in the directory "**PROGDIR:**". Shared libraries will use the startup code "**library_startup.o**," the others "**startup.o**".

Without Startup-Code

Normally only "drivers" and "handlers" are linked without Startup-Code. Here the libraries mentioned in the project will be linked, but the Startup-Code will not be used.



If this option is selected the string gadget is activated and you can enter your own object file that should be used as the Startup-Code for linking.

Own Startup-Code

Now all functions and global variables from the object files of the project modules will be linked to the program, even if they are not used. Of course this does not count for functions in libraries.

Link All Objects

StormLink creates a file which contains all symbols from the generated programs with their assigns in the finished program. Here you see part of a linker map:

Write Map File

```
_cout      = $ 24 in hunk 1  <stormc:lib/storm.lib> ( Far Public )
_std_out   = $ 32 in hunk 1  <stormc:lib/storm.lib> ( Far Public )
_std_in    = $ 16 in hunk 1  <stormc:lib/storm.lib> ( Far Public )
_cin       = $ 8  in hunk 1  <stormc:lib/storm.lib> ( Far Public )
_clog      = $ 40 in hunk 1  <stormc:lib/storm.lib> ( Far Public )
_std_err   = $ 4E in hunk 1  <stormc:lib/storm.lib> ( Far Public )
_cerr      = $ 40 in hunk 1  <stormc:lib/storm.lib> ( Far Public )
```

The first entry of a link is the name of the symbol. Then the hexadecimal value follows, then the hunk in the program where this symbol can be found, the object file it came from, and the memory class of the hunk. The map file has the same name as the generated program with the ending ".MAP".

StormLink looks for multiple occurrences of symbols. If something is found, the name of the file in which the symbol was defined will be shown. The test will be made on libraries as well, because names could be multiply used, but with different meanings.

Protocol Name Conflicts

A good example is the C function "**printf**". It is in "**MATH.LIB**" as well as "**Storm.Lib**", but the one in "**Math.Lib**" can print floating point values.

Sets the offset of the data section for the created program to the value: first data object + number. If your program contains a lot of data and StormLink gives you the error message "**16 bit Reference/Reloc Overflow**," a base number of 0x8000 could fix the problem. If this does not help, you must use 32 bit addresses (HUGE data model).

Offset Of Linker Data base

Version

This sets the global constant "**__VERSION**" to the value of the string gadget.

Revision

This sets the global constant "**__REVISION**" to the value of the string gadget.

Library Path

Load Logfile

If StormLink can not find a symbol, it prints the message "**ERROR 25 . . .**" Before that is a text which shows which file it has to load to find the symbol. The logfile which is loaded with this option contains information about object files and the there defined symbols. It has the following setup:

```
<filename> <= This is an object file
      <symbol>
<symbol>
```

If "**Load Logfile**" is active the name of the logfile can be entered into the string gadget.

Warnings

32 Bit Reloc

When the program contains "**32 Bit Reloc**" StormLink outputs a warning. This option is good for writing position independent programs.

Optimiser

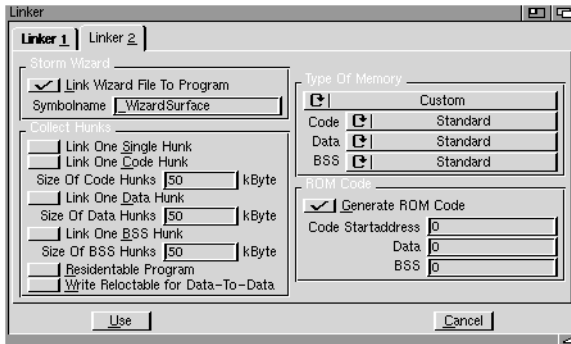
Optimise NearCode Model

When this option is set StormLink runs an optimisation pass. If it detects a 32 Bit Reference which points to the same hunk, StormLink tries to change it into a PC relative, so the reloc entry will be freed, which saves 4 Bytes.

This method may lead to incorrect programs. The optimiser only has a small disassembler, which only knows the commands **JSR**, **PEA** and **LEA**. It has no idea of how to distinguish between code and data in code hunks. The optimiser will be called for global references only, when building references between object modules. For reloc entries an assembler or a compiler should do these optimisations.



Hunk Optimisations and Memory Settings



Summarise Hunk

StormLink will only create one CODE hunk which contains all code, data and BSS sections. This is an interesting thing for game programmers.

Create Single Hunk

A reloc routine has to work on one hunk only. When the program is in SmallData model you can write completely PC relative programs in a high-level language. The base register for data access (normally A4) will be initialised with:

lea _LinkerDB(PC), a4

All other accesses are made with **d16(pc)** or **d16(a4)**, and the program will not contain any absolute addresses. Such programs can be loaded very fast and are re-entrant in most cases. In conjunction with **"Create ROM Code"** you will get a program that can be loaded to any even address by Read(file, buffer, len) and is immediately usable there. One condition is that the program does not use absolute addresses. So you should use **"32 Bit Reloc"** in any case.

A program which should be resident in memory has high requirements in the runtime system. A global variable must not be declared the way another instance of the program could use it.

Resident Programs

Principally a resident program has a small data model and it copies all data at every invocation, and works with the copy. It is important that the program does not contain any absolute addresses, because they always point to the original

data segment. These accesses can not be redirected to the copy. The option "**RESIDENT**" will switch these warnings on.

Write Data-Data-Relocation

If there are fixed 32 Bit Relocations of data to Data/BSS you can tell StormLink to create an internal relocation table for the program, so that the Startup-Code can compute the relocation for the copy of the Data/BSS hunk on its own.

Manner of Memory

You can select "**Standard**", "**Chip RAM**", "**Fast RAM**" and "**configurable**". You should use "**Standard**" so the program can choose the best settings according to the available hardware. With these settings you can force the entire program or single hunks to be loaded into Chip or Fast RAM.

ROM Code

Creating ROM Code

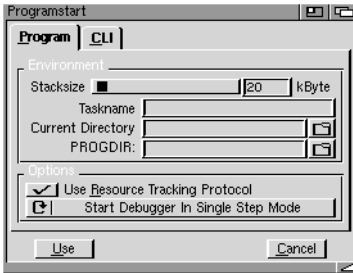


When the program uses the data relative to an address register, the BSS hunk will be assumed to be after the data hunk. The address for BSS is the address of the data + length of data. The options for SmallCode, SmallData and SmallBSS will be set automatically, so there will be no fragmentation. With the additional option "32 Bit Relocation" and corresponding Startup-Code, you can write totally PC relative programs.

When you want to burn an EPROM, you must use these settings. The output file contains pure binaries of the code and data hunks. The BSS hunk should be located at a certain address with "**AllocAbs()**". Code and data hunks are relocated to the pretended address. The AmigaDOS overhead of reloc hunks, etc. are not added. After the code there is only data in the target file.



Call of the executable program



Execute Environment

The string gadget or the slider can be used to adjust the stacksize for the program, which will be reserved by the RunShell.

Stacksize

You can enter the name of the Exec task here.

Taskname

You can enter the name of the directory that will be the current one for the program.

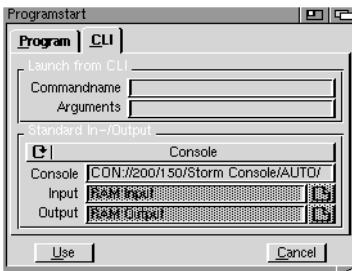
Current Directory

You can enter the name of the home directory of the program.

PROGDIR:

Start From CLI

When started from a Shell you can supply arguments which will be evaluated by C by the parameters "**argc**" and "**argv**". At program startup from within the development environment, you can simulate this manner of parameter handling.



The CLI command name is the name that is handled by "**argv[0]**".

Command Name



Arguments

Here you enter the arguments that will be given to the running program by the development system.

I/O Settings

You can choose between "**Console**" and "**Files**" for Input/Output. "**Console**" will use "**stdout**" and "**stdin**". Normally a console is opened with "**CON:**". When you want a window to be opened with special attributes you must provide further data. You can enter the starting size, the position and title of the window. The syntax is as follows:

CON:x/y/width/height/title/options

X

Number of pixels from the left border of the screen to the left border of the Shell window. When you do not enter a value (/), the lowest number of pixels will be used.

Y

Number of pixels from the upper border of the screen to the upper border of the Shell window. When you do not enter a value (/), the lowest number of pixels will be used.

Width

Width of the Shell window. When you do not enter a value (/), the lowest number of pixels will be used.

Height

Height of the Shell window. When you do not enter a value (/), the lowest number of pixels will be used.

Title

This defines the Shell window title.

The parameters of the options must be separated by a slash.

AUTO

The window will be opened automatically if the program needs data or wants to provide output. After the Shell window is open you can enter data immediately.

ALT

When you click the zoom gadget the window will be resized and repositioned according to the values given.

BACKDROP

This window will be located behind all Workbench windows. You can not put it in the foreground, so you must change all other windows to have a look at this one.

CLOSE

The window will get all standard symbols including the close gadget.



The window will be opened, but it will not become the active one.

INACTIVE

The window will not have a left or bottom borders. There will only be zoom, fore/background and close gadgets.

NOBORDER

The window will be opened without a close gadget. When console window is opened with AUTO there will be a close gadget automatically added.

NOCLOSE

The window has no fore/background gadget.

NODEPTH

The window can not be moved. There is zoom and fore/background gadget, but no drag gadget.

NODRAG

The window has a fore/background gadget only.

NOSIZE

The window will be opened on a public screen which must already exist. You must enter the name of the public screen after the key-word SCREEN.

SCREEN

When you enlarge window size the new area will be filled with text so you will have a look at parts that were out of the visible area before.

SIMPLE

When you enlarge window size the new area will NOT be filled with text. This saves memory.

SMART

The window can only be closed with the close gadget or by pressing <Ctrl>+<\>. If WAIT is the only option no close gadget will be available.

WAIT

Input/Output

When you select "**Files**" on the upper cycle gadget the string gadgets "**Output**" and "**Input**" become available. You can enter the path and the name of the file where all output data should be stored at "**Output**" and the corresponding ones at "**Input**".

Save Project

After you have made all your project settings, you can use them for your current project by clicking "**Use**," or pressing <U>.



You can now save these settings permanently by clicking the **"Save Project"** icon on the toolbar or with the menu item **"Project/Save"**

Build your own Project Template

A project template is a default project that is used when a new project is created. It is stored as **"template.1"** in the directory **"PROGDIR:"**. If you want to build your own project template you should open a new project and make your individual settings. To save this project as your project template you must select the menu item **"Project/Save As Project Template"**.



StormED has some special features that makes writing source more comfortable and easy. StormED will be used every time the source is displayed. This is mostly when editing your text, but if you are debugging, your source will be shown in the editor as well. This will make it easier for you to become familiar with the controls of StormC, because they are always the same. You will have worked through "First Steps" by now, so you are more familiar with StormED. If you are working with it often you will come to appreciate the syntax highlighting because it enable you to recognise errors very quickly. The following chapter will tell you everything about StormED. You will see it is a wonderful tool to use.

IN GENERAL	93
New text	93
Controls of windows	93
Open / Load Text	94
<i>Tooltypes</i>	94
Save Text / Save Text As ...	94
Free Text	94
Options of Text	95
<i>Tabulations and Indents</i>	95
<i>Indent ahead and behind of brackets</i>	96
<i>Dictionaries and Syntax</i>	97
<i>Dictionaries</i>	97
<i>Syntax</i>	98
<i>Colour Settings</i>	99
<i>File Saving Settings</i>	99
<i>Saving the Settings</i>	100
Keyboard Navigation	100
<i>Cursor-Keys</i>	100
<i><Return>, <Enter>, <Tab></i>	100
Undo / Redo	100
Block Operations	101
<i>Mark, Cut, Copy and Paste</i>	101
The Mouse	102



Find and Replace 102

Direction 103

Mode 103

Ignore Upper/Lower Case 103

Ignore Accent 103

Find 103

Replace 104

Replace All 104

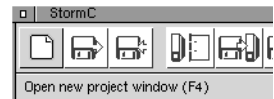


IN GENERAL

Before you start with this section you should have learned a little about the project manager. You will already know that normally you do not open a text without opening a project first. Opening "pure" text is only recommended if you want to have a quick look at it. All sources that belong to a project should be contained within it. Opening text will then be easier, because you can double click on it's entry in the project manager.

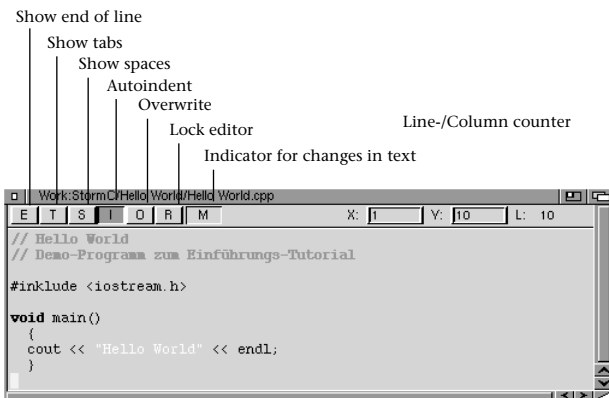
New text

Click on the icon **"New Text"** in the menu bar to open a new text window You can now start typing. As mentioned before, this is not the usual way to work. You should open a new project first and then add new text with the menu item **"Project/Add File(s)..."**. You will find more information on this in the chapter about **"Project Manager"**.



Controls of windows

You will notice that there is a row of letters at the top line of the editor window.



If you move the mouse pointer over the (text) icons you will get an appropriate explanation in the information line at the top of the editor window. In the right top corner of the editor window you will find the line and column counters. They are active for input and output. So if you want to move to a certain line in your source simply click into the line counter string gadget and type the line number you want to

go to. As soon as you hit <Return> the cursor will jump to the new position (if you have chosen an existing line number!).

Open / Load Text

You will probably have wondered about the difference between opening and loading text.



The menu item **"Load"** is only available when an editor window is active. The text will be loaded into this editor window.

If you **"Open"** text this will open a new editor window.

Tooltypes

All attributes of texts (like text colour, tab size, bracket indent) and the size of windows are stored in the tool types of the respective icon of this particular kind of text. They will be applied when opening and loading. At loading the size of the window will not be recognised.

Save Text / Save Text As ...

If you create new text it will be named **"unnamed"**. This appears in the title bar of the editor window. If you made any input to this text **"Save"** and **"Save As..."** will have the same effect. The ASL filerequester will be used for selection of a filename. Unfortunately you cannot double-click to make this selection because ASL does not support this. If you stored the text once it will be overwritten every time you choose **"Save"**. The tool type (window size, text attributes) will be adjusted as well.



Free Text

To remove the text in your active editor window you can dispose of it's contents. Choose the menu item **"Project/Clear"**. This item is only available if there is an active editor window. The text will be disposed of but the window will stay open so you can load another one or start typing a new one. When text has not been saved before you will be asked to do so or to accept the disposal.

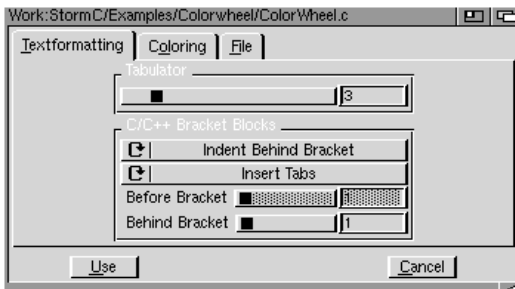
Options of Text

Before explaining further functions of StormED you will need to know more about the options of the editor. The following sections will always refer to these options, because most of the functionality of StormED depends upon it. The chapter "Project Manager" has already explained the global options of the editor so here we will only discuss the text options.

To get the options requester you must activate an editor window and choose the menu item **"Settings/Text..."**. A requester will appear on the screen similar to the one of the project options.

At the top of the requester you will see paging gadgets with the following titles:

Tabulations and Indents



You can change the width of the tab stops with the slider or you can type its value into the string gadget. Values between 1 and 16 are allowed.

Tabulation

Most C programmers use a special structuring of their source to get a better overview. There is no standard instruction for this and so there are many possibilities of how this can be achieved. We therefore leave this up to you. You can configure the options for some automation to make structuring easier.

C/C++ Bracket Blocks

When you do not want the editor to affect the structuring of your texts please select **"No Indent"** at the first cycle gadget. Then there will be no indentation after you typed a bracket. On any other selection the cursor will be indented an adjustable amount of characters. You can choose if this should happen when typing the bracket or after you hit

<Return> or <Enter>. The values of **"Ahead of Brackets"** and **"Behind of Brackets"** will affect the amount of characters ahead of or behind the brackets. You can adjust the values with sliders or by typing into the string gadget. The range is from 1 to 16.

The value does not always mean single characters because it is effected by the context of the next cycle gadget. If it says **"Indent Spaces"** it will be real characters. But if it says **"Insert Tabs"** the cursor takes over the value of **"Tab Size"**. For example: when you set **"Tab Size"** to 8 and **"Ahead of Brackets"** to 4 the cursor will indent 32 characters and insert a bracket there. If you choose **"Insert Spaces"** the cursor would be moved 4 characters to the right and the bracket would be inserted here.

The use of automatic indent takes effect only if you activated the indent function of the editor. Otherwise the cursor will be placed in the first column after hitting <Return> or <Enter>.

Indent Brackets

When you set the option **"Indent Brackets"** a bracket will be set according to the value set at **"Ahead of Brackets"**. An amount of spaces or tabulators will be inserted in front of the bracket and the cursor will be placed behind it.

Indent behind of Brackets

To activate automatic indent according to **"Indent behind a Bracket"** you must hit <Return> or <Enter> behind a bracket. According to the value of **"Behind of Brackets"** the cursor will be moved a certain amount of spaces or tabulators to the right. This will be done relative to the position of the cursor.

For example: when the cursor is in column 10 and you type a bracket the cursor will be placed in the next line in column 12 (if you set **"Insert Spaces"** and **"Ahead of Bracket"** to 2).



Ensure that the Indent function is active for these settings to take effect. You can check the status by the "I" icon.

Indent ahead and behind of brackets

The option **"Indent ahead and behind of Brackets"** is a combination of **"Indent Brackets"** and **"Indent behind of Brackets"**. If you type a bracket a certain amount of spaces or tabulators will be inserted. The bracket will be set and the cursor will be placed behind it. If you enter <Return> or <Enter> afterwards the cursor will be placed in the next line and it will be moved the indicated amount



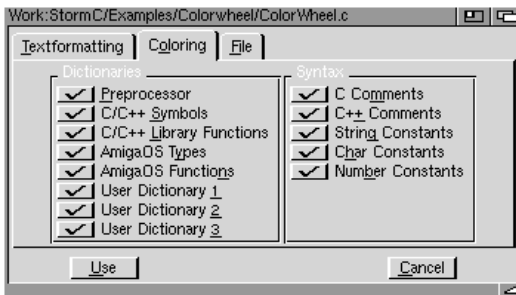
of spaces or tabulator to the right. If you enter one or more characters instead of `<Return>` or `<Enter>` the function will be terminated and after the next input of `<Return>` or `<Enter>` the cursor will be placed in the following line just under the bracket.

If all these explanations seem to be confusing, please take some time to use the functions yourself. This way you will see that all these settings are actually very simple in use.

Dictionaries and Syntax

When you chose this settings page you will see two lists of gadgets. The first group is called "Dictionaries" the second "Syntax".

Text colouring



Dictionaries

In this group you will find a list of gadgets to control the use of dictionaries for text colouring. The dictionaries are stored in the drawer **"StormC:StormSYS/Dictionary"** and you can load them with the editor.

Pre-processor

"Preprozessor.dic"

Here you will find all the defines of constants, that means everything that starts with **"#define"**.

C/C++ Symbols

"C Symbols.dic"

This dictionary contains the C type-definitions like **"auto"**, **"sizeof"**, ...

C/C++ Libraries

"C Library.dic"

This includes all ANSI C and C++ standard functions.

AmigaOS Types

"Amiga Types.dic"

In this dictionary you will find the Amiga specific types like **"ULONG"**, **"STRPTR"**, ...

AmigaOS Functions

"Amiga Functions.dic"

This dictionary contains a list of all names of AmigaOS functions.

The following tree entries are for user-defined dictionaries. They do not contain any settings. The files are named **"User 1.dic"**, **"User 2.dic"** and **"User 3.dic"**.

If you activate the functions here the words of the texts in the editor will be marked in various colours according to the colour definitions of the dictionaries.

Syntax

On the right side of the settings page there are controls for further colouring of the editor.

C Comments

A comment in C begins with **"/*"** and ends with **"*/"**. Comments might be longer than one row, but they must not contain further comments. As soon as you type **"/*"** the following text will be coloured. This will end when you enter **"*/"**.

C++ Comments

C++ comments are finite to one row. It will be marked with **"//"**. There is no marker for the end of such a comment.

String Constants

The marking of a string constant will start with a character you enter with **<Shift>+<2>**. It will be ended with the same character.

Character Constants

Character constants will be marked with a single **"'"** (**<Shift>+<Ä>**).

Number Constants

All numbers that are not contained within quotation marks or semi-colon will be coloured by this function.



Colour Settings

The settings of colours will be made in the file **"StormSettings.ED"**.

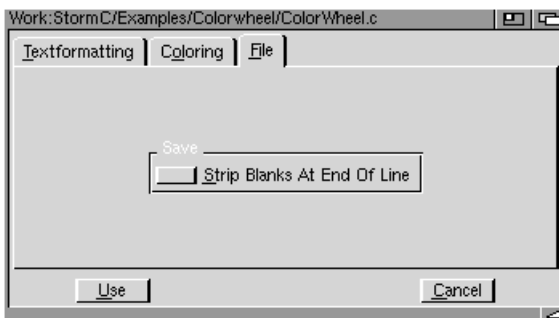
For example: ...

Only the lower 5 rows of this settings file are of interest. The definition has the following setup:

1. **foreground colour (RGB)**
2. **background colour (RGB)**
3. **foreground index**
4. **background index**
5. **softstyle**

The separation into these values has the advantage that AmigaOS 3.0 and above can use palette sharing. The OS function will look to see whether there is the requested (or other suitable) colour it can use. AmigaOS 2.1 and older will use fixed colour indexes. That is the reason for both methods used here. If you use AmigaOS 2.0 or older the colour used depends on the colour settings you made on Workbench. AmigaOS 3.0 and above will try to get the colour you requested or a colour that is close to this one. The result might not be as good as you expect, but this is a common problem on computers with only a few colours available.

File Saving Settings



To prevent the saving of text with spaces at the end of some lines you can turn on the function **"Strip Blanks At End Of Line"**.

Saving the Settings

All the settings you made will be saved as Tool Types of your text. If you want these settings to take effect on every new text you create you must activate the menu item **"Project/Save as Default Icon"**.

Keyboard Navigation

Most of the actions of the editor can be activated by special keys. The following paragraphs will give you an overview about the keyboard navigation.

Cursor-Keys

You can move the cursor by the use of the cursor-keys and the regular combinations with `<Alt>`- and `<Shift>`-keys.

`<Alt>`+*cursor left / right* will move it to the beginning of the word before or after the current cursor position. `<Alt>`+*cursor up / down* will move to the beginning or end of the text.

`<Shift>`+*cursor left / right* will move to the start or end of the line. `<Shift>`+*cursor up /down* will turn over one page up or down.

<Return>, <Enter>, <Tab>

When you hit `<Return>` and the cursor stands in a text line, the line will be split. When you enter `<Return>` at the beginning of a line a spaced line will be inserted before this line. When you enter `<Return>` at the end of a line a spaced line will be inserted after this line.

These actions will be the same when you enter `<Enter>` instead of `<Return>`.

Using the `<Tab>`-key will insert 3 spaces or the value you set in the text settings. Using Tabulators will improve the structure of your source so you can read it better.

Undo / Redo

One of the most important functions of a text editor is the possibility to undo other functions like editing or deleting text. Redo is the reverse function of Undo. So if you typed a text and entered "Undo" it disappears. When entering "Redo" it will appear again. StormED offers unlimited Undo/Redo.



For example:

Type the following small program into the editor window:

```
while (!sleep)
    sheep++;
```

Hit the keys <Right Amiga>+<Z> (this is the same as the menu item **"Edit/Undo"**). The text will disappear character by character. Now hit <Right Amiga>+<T> (this is the same as the menu item **"Edit/Redo"**). The text will appear again.

Block Operations

With the help of block operations you can move or copy small or large parts of your text. These functions are very important in an editor. A block is an area of the text that is marked. The fundamental block operations are ...

Mark, Cut, Copy and Paste

To execute a block operation you must first mark a block. Choose the menu item **"Edit/Mark"** and move the cursor. The area between the starting point of the cursor and its current position will be inverted. This shows you that this area is marked. All block actions will effect this area only. If you choose the menu item **"Edit/Mark"** again the marking of the text will disappear.

"Cut" will cut out the marked block of the text and copy it to the clipboard. The block is no longer visible. **"Copy"** will copy the marked block to the clipboard. It will remain visible. **"Paste"** inserts the contents of the clipboard at the current position of the cursor into the text.

For example:

- Please load a large text file into the editor window.
- Move the cursor to the middle of the first line of the text and choose menu item **"Edit/Mark"**.
- Move the cursor down a few lines. You will see an inverted area of your text.
- Choose menu item **"Edit/Cut"** and the marked block will disappear. The rest of your text will be moved to fill the free area.

To restore the old text you must paste the clipped block from the clipboard to your text. But remember that you should

not move the cursor, because it marks the position for the insertion of the block. So if you move it, the block will be inserted in a different place.

- Now choose menu item **"Edit/Paste"**. The block will be inserted from the clipboard and your text will return. The cursor will be at the end of the block.

To insert the block at a different place within your text, move the cursor to the desired position and choose **"Edit/Paste"** again.

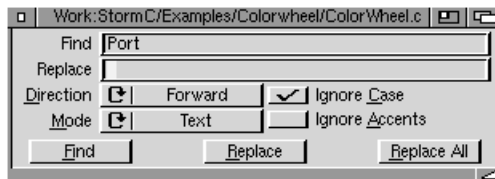
When you want to copy a certain block of your text, just mark it and choose **"Edit/Copy"**. The text is not affected, but it is copied to the clipboard.

The Mouse

Using the mouse you can move the cursor to any position of the text. Just move it to the right place and hit the left button of the mouse. As long as you hold the left button down the cursor will follow your mouse-movement and the text will be marked and inverted. It is the same as using the menu item **"Edit/Mark"**. When you move the mouse cursor to the top of the window while pressing the left button, the text will be scrolled in this direction. After you release the left button of the mouse the block will still be marked until you click it once more.

Find and Replace

Find and replace is one of the basics features of any editor. You will get a requester when you choose the menu item **"Edit/Find&Replace"**.



Type the word you are looking for into the first string gadget. If you don't want to replace it, leave the next string gadget unfilled and hit the button **"Find"**.



Direction

You can choose the direction for the search from within a cycle gadget. You can search "**Forwards**", "**Backwards**" and "**Whole Text**". "**Forwards**" and "**Backwards**" will start from the current position of the cursor. "**Whole Text**" does not care about the cursor position and looks for your key-word through the whole text.

Mode

This option defines how the search word is compared against the text.

The key-word will be searched throughout the text. A key-word like "**port**" would be found in words like "**userport**" and "**ports**".

Text

This would only find words that exactly match the key-word.

Whole Word

This would look for words which begin with the key-word. So "**ports**" would be found, but not "**userport**".

From Beginning of the Word

This would look for words which end with the key-word. "**Userport**" would be found, but not "ports".

From the End of the Word

This would only look at the first word of a row.

From Start of Line

This would look at the end of the row only.

From End of Line

Ignore Upper/Lower Case

This would ignore whether the word is in upper or lower case.

Ignore Accent

This would ignore whether a word contains accents or not. Accents would be treated as they were normal letters. E.g. "RenÉ" and "Rene" would be the same.

Find

This starts the search process. If there is no matching word you will hear a "display beep".

Replace

If you choose "**Replace**" the function will look at the first match of text and key-word. Then it would replace this matching text with the key-word and continue the search.

Replace All

This would do the same as "**Replace**", but it would not stop until it has searched (and replaced) the whole text.

The "**Find&Replace**" requester will stay open as long as you are working with it. This is a non-modal requester. You can always close it with the close gadget or by hitting <ESC>-key.



Despite computers are getting faster and faster you do not want to forego on a fast compiler. This does not only mean speed at compiling but speed, which benefits through used optimisation to speed up programs.

StormC offers both in an excellent way. It is a very fast compiler with outstanding optimisation. C++ sources will be compiled to machines-code in one pass. Pre-compiled header care for tremendous compiler speed at large projects.

You will learn more about special features of StormC in this chapter.

SPECIAL FEATURES OF STORMC	107
DATA in Registers	107
Parameters in Registers	108
Inline Functions	108
The Pragma instructions	110
<i>Data in Chip and Fast RAM</i>	<i>110</i>
<i>AmigaOS Calls</i>	<i>111</i>
<i>The #pragma tagcall</i>	<i>111</i>
The #pragma priority	112
<i>Constructors and Destructors in ANSI C</i>	<i>113</i>
<i>Constructors and Destructors in C++</i>	<i>113</i>
<i>Priority list</i>	<i>113</i>
Joining Lines	114
Predefined symbols	115
Build your own INIT_ and EXIT_ routines	117
Use of Shared libraries	117
<i>Prototypes</i>	<i>117</i>
<i>Stub Functions</i>	<i>118</i>
<i>#pragma amicall</i>	<i>118</i>
<i>Forced opening of a Amiga library</i>	<i>122</i>
PROGRAMMING OF SHARED LIBRARIES	123
The Setup of a Shared Library	123
<i>The #pragma Libbase</i>	<i>124</i>
<i>Register Setup</i>	<i>124</i>

<i>Shared Library Project Settings</i>	125
<i>The setup of FD files</i>	125
<i>The first four Functions</i>	126
<i>Home-made Initialisation and Release</i>	127
<i>Important hints to Shared libraries</i>	128
PORTING FROM SAS/C TO STORMC	130
Project settings	130
Syntax	131
Keywords	131
CLI VERSION OF THE COMPILER	134
The instruction	134
Options	134
<i>Assembler source</i>	136
<i>Pre-processor: Definition of symbols</i>	136
<i>Pre-processor: Include files</i>	137
Compiler mode	137
<i>ANSI C or C++</i>	137
<i>Exception handling</i>	137
<i>Creation of Template functions</i>	138
Code creation	138
<i>Data model</i>	138
<i>Code model</i>	138
<i>Optimisations</i>	138
<i>Code for special processor</i>	140
<i>Code for linker libraries</i>	141
Debugger	141
<i>RunShell</i>	141
<i>Symbolic debugger</i>	141
Copyrights	141
Warnings and errors	141
<i>Format of the error output</i>	141
<i>Colours and styles</i>	142
<i>Error file</i>	142
<i>Optional warnings</i>	143
<i>Treat warnings like errors</i>	144
<i>Core memories</i>	144
<i>Pre-compiled header files</i>	144
Summary	145



SPECIAL FEATURES OF STORMC

DATA in Registers

Each processor manages not only the RAM, but has in addition also some internal processor registers, in which it stores results of calculations and so on. The access to these registers is considerably more quicker than a RAM operation. Since the processors of the 68000-family are richly equipped with registers programmers will take the unused registers to archive often used variables into them to increase speed.

Fortunately you have introduced a flexible and machine-independent notation for it in the ANSI C standard. The key is a memory class named **"register"**, which can be used analogue to **"extern"**, **"auto"** or **"static"**.

Register variables may be declared only within functions. Otherwise you can use them freely: You may declare as many variables as you want from different types as **"register"**, the compiler decides later on which files to store in a register. Integer and pointer or reference variables may be put into registers, whereby numeric types are stored in data and pointer types in address registers.

However you should be careful not to use registers too extensively because the processor will need at least one register for practically all operations. When you are using too many of them for variables it must evacuate them sometimes for a short time on the stack and load them later again. Then you can not expect an increase in speed. As a rule you should not use more than four integer and two pointer variables as **"register"** in one function.

StormC does not use your statements obligatorily. Indeed it takes them only as reference. The optimiser of StormC can decide essentially better, which variable to which time in which register is to be loaded best. Furthermore StormC uses the registers repeatedly in a function, which you can not make by using the key-word **"register"**.



*This does not
corresponds to
C++ standard!*

*You should insert this
feature therefore as
carefully as possible, since
the programs are otherwise
no more portable. It is best
to use such parameters only,
if you want to bind finished
functions, which expect
their arguments in registers,
to a program. E.g. the
AmigaOS functions or
assembler routines, which
can be ported badly
anyway.*

Parameters in Registers

As small expansion of the C++ standard StormC offers the possibility to also submit parameters in registers. Sometimes this leads to certain speed increases. You simply have to write the symbol **"register"** before the parameters of the declaration of the function, e.g.

```
int Inc (register int i, register int j);
```

There are even two possibilities to prescribe the compiler in which registers it should put the parameters: Either you rename the parameter variable as a 68000 register, e.g.

```
void f(register int d0, register char *a2)...
```

or you set the desired register name with **"__"** behind the key-word **"register"**

```
void f(register __d0 int i, register __a2 char s)...
```

If you are instructing the compiler this way which registers it should use for variables you should consider previously about the things you are doing there.

For your own C++ functions this feature is absolutely taboo, because you don't know whether the above declared function would become possibly very slow because it needs the register **"d0"** constantly but it must store and restore the content of the registers variable every time. You should use such register regularise for declaration of imported assembler and system functions only.

Inline Functions

Each function call represents a considerable expense. First the calling code must arrange some space for the arguments on the stack and store them there. Then a **"JSR"** jump follows into the function, which has to reserve the necessary memory for their own data on the stack before. And it has to store the contents of all processor registers, which are changed by it. At the end of the function the old register contents are restored again. The local variables are removed from the stack; a **"RTS"** jump back to the calling program is following. At least the arguments must be taken from the stack. There you as programmers might get into moral conflicts: Should you define a small function now or insert



the content of this function every time into the program? A typical example is the maximum function:

```
int max(int a, int b)
{ if (a>b)
    return a;
  else
    return b;
}
```

Instead of the function call "**max(x,y)**" you could as an alternative insert "**(x>y ? x : y)**" each time. This would be of course more quicker, but not more readable. For such cases C++ has a nice feature that StormC now offers in ANSI C mode as well: "Inline" specification.

If you set the key-word "**inline**" (in ANSI-C "**__inline**") before the first declaration of a function, this function definition will be inserted into the source at each time this function is called. Syntactically or semantically the program is not changed thereby: You declare simply

```
inline int max(int a, int b)
{ if (a>b)
    return a;
  else
    return b;
}
```

Then you can use "**max**" like a normal function. The difference lies solely in the generated code. With "**inline**" the content of "**max**" will be copied every time into the calling code.

A lot will be saved thereby: There must neither be a "**JSR**" into the function nor a "**RTS**" back. There must be no storing and restoring of the contents of registers. The function must not arrange its own data "**frame**" on the stack. The result not given back in register "**d0**". * And the compiler will still have many possibilities for optimisations.

An inline function is not created once only, but in many copies, for each call a separate one. It is obvious that "**inlining**" will blow up programs tremendously and you should use it carefully. The compiler however may ignore that "**inline**".

StormC handles this in a better way: what you want to be **"inlined"** will be **"inlined"**. Still there can be situations, in which an inline must be created on completely usual manner, e.g. if you take the address of a such function if it is used already before it was declared if an inline function is recursive.

All this is allowed because you can use an inline function exactly as each other function.

If an inline function is created for one of the above reasons, it naturally has internal linkage. So other modules will not notice whether the function was actually generated in a compiler. Therefore similar instructions like for the memory class **"static"** will apply for **"inline"**. The declaration sequence

```
inline void f(); void f();
```

is allowed,

```
void f();  
inline void f();// ERROR
```

is not allowed. By the way inline is not a memory class (like **"static"** or **"external"**), but a **"function specification"**.

The Pragma instructions

Despite the ANSI specifications each compiler has its own peculiarities. These specific things are introduced with the key-word **"#pragma"**.

"#pragma" lines will be interpreted by the pre-processor like **"#include"** lines. With **"#pragma"** you can realise compiler specific functions, which are not standardised explicitly.

Compiler mode **"#pragma -"** is a non-standard feature switches StormC into ANSI C mode.

"#pragma +" switches the compiler to translate the source in the C++ mode.

Data in Chip and Fast RAM

The architecture of the Amiga is somewhat unorthodox. So there are two or sometimes even more different types of



RAM. Normally programmers are only interested in using "**Chip- or Fast-RAM**" because they must care for that graphic data are stored in chip memory. StormC therefore offer the pragmas "**chip**" and "**fast**".

All after the line

```
#pragma chip
```

declared static datas will be loaded into Chip RAM,

```
#pragma fast
```

switches back to normal mode. Now data will be stored somewhere, however Fast RAM is the preferred place.

AmigaOS Calls

The AmigaOS functions are called with the `#pragma amicall`. This declaration exists in the essential of four parts:

- The name of the basis variable.
- The offset as positive integer.
- The function name, which must be declared already. For reasons of the definiteness this function name may not be overloaded.
- The parameter list, represent through a corresponding amount of register names in parenthesis.

An example:

```
#pragma amicall(Sysbase, 0x11a, AddTask(a1,a2,a3))  
#pragma amicall(Sysbase, 0x120, RemTask(a1))  
#pragma amicall(Sysbase, 0x126, FindTask(a1))
```

Normally you will never have to write such declarations by your own, since everything is included within the Amiga-Libraries.

The #pragma tagcall

At some OS calls so-called TAG lists will be submitted as function parameters. These are not really OS functions but STUB functions, which are declared with ellipse. You are using these functions, since the submission of TAGs as function parameters is to be programmed very simply. The use of an array would be of course be a good way too, but it causes more work on typing.

An example:

The pragma definition of the function "CreatGadgetA()" from the Gadtools-Library looks as follows:

```
#pragma amicall(GadToolsBase,  
               0x1e,CreateGadgetA(d0,a0,a1,a2))  
#pragma tagcall(GadToolsBase,  
               0x1e,CreateGadget(d0,a0,a1,a2))
```

You find the corresponding prototype in the drawer "CLIB". They contain the following:

```
struct Gadget *CreateGadgetA(unsigned long kind,  
                             struct Gadget *gad, struct NewGadget *ng,  
                             struct TagItem *taglist);  
struct Gadget *CreateGadget(unsigned long kind,  
                             struct Gadget *gad, struct NewGadget *ng,  
                             Day tag1, ...);
```

With the call

```
gad = CreateGadget(CYCLE_KIND, gad, ng,  
                  GTCY_Labels, mlabel,  
                  GTCY_Active, 3,  
                  TAG_DONE);
```

you would "so to speak" inline create a Stub function, which pushes all Tags on the stack but in reality calls the function "CreatGadgetA()".

For compatibility reasons the most pragma calls of the SAS compiler can be used too.

The #pragma priority

StormC supports as a C++ compiler the automatic call of initialisation and exit functions before and after call of "main()" function.



*Whether stdio
will be initialised
or not will not
depend on startup-code,
but if the program uses
printf() or similar I/O
functions.*

C++ uses this feature to call the constructors and destructors of global variables. ANSI C uses this feature for the automatic opening of shared libraries or for the initialisation of standard I/O.

The function table, which is processed thereby, is created first through the linker. Functions, which own a certain name, will be gathered by the linker to a table of



initialisation and exit functions. The startup-code must call the function **"InitModules()"** before the call of **"main()"** function and to call **"CleanupModules()"** after **"main()"** function.

The advantage of this dynamic concept is the flexible setup of the startup-code. The resulting program is always as small as possible. The thereby available startup-code is applicable for all programming projects. The large-scale and error-susceptible selection of a startup-code according to compiler options is not necessary any more.

Constructors and Destructors in ANSI C

Functions that are called at initialisation must be named **"INIT_n_anything(void)"**, the exit function must be called **"EXIT_n_anything(void)"**. For *n* is a number between 0 and 9. This sets the priority of the function. The smaller the value the earlier the initialisation and the later the exit function will be called. **"Anything"** can be replaced through each arbitrary name, which may contain further **"_"**.

Constructors and Destructors in C++

A C++ compiler creates automatically such fitting functions for each global variable. In these functions the construction and destruction of variables will take place. Generally the compiler uses the priority 8 for these **"INIT_"** and **"EXIT_"** functions.

However the exact sequence of the calls, that means the priority of the **"INIT_"** and **"EXIT_"** functions, may be of importance so there exists that **"#pragma priority"**. You can determine the priority of the global variable of the module.

Priority list

Thereby you should orient yourself on the following list of the priority. This tells you the use of the priorities in **"storm.lib"**:

This priority is reserved for functions, which initialise the StormC library (particularly data structures of the library). If you want to initialise a data structure (e.g. a jump table), that does not access to any OS resources, you should use priority 0 also.

Priority 0:

- Priority 1:** The most important shared libraries are opened: **"dos.library"** version 37, **"utility.library"** version 37.
- Priority 2:** All shared libraries, which are necessary for the function of the program, are opened. The program is aborted, if the opening fails. Here all libraries are listed, which are delivered since AmigaOS V2.04 (V37).
- Priority 3:** Here all libraries are listed which are delivered since AmigaOS V38 or later or these that do not belong to standard delivery of the system. You do not know whether these libraries exist the concrete system. These libraries are opened with the minimal version. If the opening should fail it outputs no error message.
- Priority 4:** With **"malloc()"** or **"new"** reserved memory will be freed.
- Priority 5:** Further library resources are reserved and freed, e.g. global and temporary files.
- Priority 6:** There are some cleanups before the resource are freed, e.g. flushing of file buffers etc.
- Priority 7:** Unused, free for usage.
- Priority 8:** Constructors and Destructors C++ cases are set to this priority by default. Each global variable, whose type has a Constructor or a Destructor, creates a call of an **"INIT_"** or **"EXIT_"** function. The priority can be changed by **"#pragma priority"**.
- Priority 9:** The functions, which were announced with **"atexit()"**, are called.

Joining Lines

It is absolutely alike in C and C++ where individual lines end. But the pre-processor must get each instruction exactly in one line. Naturally there will be the case that a line becomes very long, e.g. through an extensive macro definition. For this case there is the backslash. If there is a **"\"** at the end of a line this is joined with the following one.

For example:

```
#define SOMETHING \  
47081115
```



This is a valid macro definition, for "47081115" is pulled here in the preceding line.

Predefined symbols

The pre-processor has many predefined macros. Some of them are ANSI C standard others are elements of C++ or particular peculiarities of StormC. These macros can not be redefined.

__COMPMODE__

It is defines with the "**int**" constant "0" in C mode and "1" in C++ mode.

__cplusplus

StormC defines the macros "**__STDC__**" in C and C++ mode. If you wants to check, whether if compiling in StormC is done in C++ mode this must be made with the macro "**__cplusplus**".

__DATE__

The macro "**__DATE__**" delivers the date of the compilation. This is very useful, if you want to give a program an unique version number:

```
#include <iostream.h>
```

```
void main()
{ cout << "version 1.1 from that " __DATE__,
  "__TIME__" clock\n"; }
```

The date has the format "month day year", e.g. "Mar 18 1997" and time of day will be "hour:minute:second".

__FILE__

This macros contains the name of the current source file as a string, e.g.

```
#include <stream.h>
void main()
{ cout << "That stands in line " << __LINE__ << " in
file " __FILE__ ".\n"; }
```

The value of the macro "**__FILE__**" is a constant string and can be joined with other strings standing before or behind it.

__LINE__

The macros "**__LINE__**" delivers the line number, in which it is used, as decimal "**int**" constant.

__FUNC__

The macro "**__FUNC__**" delivers the identifiers of the translated function (Aztec compatible) in ANSI C mode. In C++ mode it delivers the qualified function name along with the parameter list.

__STDC__

This macros delivers in all ANSI C compatible implementations the numerical value 1. This should tell you whether the compiler supports this standard. Otherwise "**__STDC__**" is not defined.

__STORM__

You may want to know which compiler and version one is using. Therefore StormC defines the macros named "**__STORM__**".

__TIME__

(see "**__DATE__**")



Build your own INIT_ and EXIT_ routines

If you are writing your own **"INIT_"**- and **"EXIT_"** functions you should be aware that a call of the function **"exit()"** from an **"INIT_"** function will call all **"EXIT_"** functions.

There is no allocation between **"INIT_"** and **"EXIT_"** functions, this means that there is no **"INIT_"** function that belongs to a certain **"EXIT_"** function with the same name. That applies for each global variable in C++ also. That means, that a destructor must work well even if the matching constructor was not processed. You can exploit thereby, that the memory of the variable is initialised always with 0, if you have not changed initialisation. Now you can decide simply, whether the constructor of the destructor was processed already or not.

Particularly in C++ you should avoid that constructors of global variables do an exception because they can not be enclosed with a Try-block. Therefore this exception is always treated as unexpected exception, which leads to a hard interrupt of the program and the user will not know why the program does not work.

Use of Shared libraries

The AmigaOS is essentially built of several shared libraries. These "divided libraries" offer functions which can be called by each program after opening them.

To each shared library of the AmigaOS there are some C headers, which allow the use of the library.

Prototypes

First there is the file with the prototypes of the functions. You will find them in the directory **"StormC:include/clib"**. The files are named like **"exec_protos.h"**, that is the name of the library (here: **"exec"** for the library **"exec.library"**) with appended **"_protos.h"**.

These prototypes define C functions, which are always called by the compiler in the way that it puts the parameters of the function on the stack and makes a subroutine call at the matching function. This function must always be linked to the program. Therefore it is not allowed to call the shared library directly because shared libraries are not linked permanently to the program, but are opened at the start of

the program. Besides functions of shared libraries always expect their parameters in CPU registers and not on the stack.

This problem can be solved on two ways. Either you use Stub functions or the special Amiga-like **"#pragma amicall"**.

Stub Functions

The linker library **"amiga.lib"** contains a so-called Stub function for every function of the OS libraries. The stub function takes the parameters from the stack, loads them into the right CPU registers and then calls the right function of the shared library.

This method has the disadvantage of an additional function call, which swells your program and decreases the execution speed strongly.

#pragma amicall

Alternative to stub functions you can use **"#pragma amicall"** which differs for every compiler. There are the following two methods:

There is a file to each library which describes the calling of each function of a shared library. Which parameter must be written into which CPU register is explained particularly. The compiler now can call the shared library functions directly without the use of **"amiga.lib"**.



To get a good comparability with SAS/C and other compilers there exists a drawer named "StormC:include/pragmas". It contains files which have different file names than these of "StormC:Include/pragmas".

These files can be found in the directory "StormC:include/pragmas" and they are named like "exec_lib.h" (name of the library with appended "_lib.h"). These files will load the prototypes from "StormC:include/clib" automatically, so one "#include" of the pragmas will be enough.

You can tell the prototypes to load pragmas as well. That is practical, if they have an older source, which uses only the prototype and contains no includes for pragmas. For this the pre-processor symbol **"STORMPRAGMAS"** must be defined. You can define the symbol simply in the compiler settings on **"Pre-processor"**.



Furthermore nearly every library has at least one include file which describes the data structures which are required for the library.

Example:

```
#include <pragma/exec_lib.h>
#include <exec/tasks.h>
#include <stdio.h>

void main()
{
    struct task *me = FindTask(NULL);
    printf("The structure has address %p.\n", me);
}
```

This program uses the function `"FindTask()"` of `"exec.library"` to get a pointer to the control structure of the task and it outputs their memory address.

The `"exec.library"` has a particularity to all other libraries. It will not be opened by the program, because it is always open and the functions can be used immediately.

This is different if you have a look at e.g. `"intuition.library"`.

Example:

```
#include <pragma/exec_lib.h>
#include <pragma/intuition_lib.h>
#include <exec/libraries.h>

external struct Library *intuition base;

void main()
{
    if((intuition base =
        OpenLibrary("intuition.library", 37)) != NULL)
    {
        DisplayBeep(NULL);
        CloseLibrary(intuition base);
    };
};
```

This program makes nothing than flashing the screen. Maybe it will play the 9th Symphony of Beethoven too if you have reconfigured system beep.



In this context one says "pragma files" or "the compiler XYZ knows Pragas". That is very inaccurate, for the instruction `#pragma` has nothing to do with the call of shared libraries, for it is ANSI C standard. Indeed `"#pragma amicall"` is very important and older compilers on the Amiga knew this `"#pragma"` only. Therefore `"#pragma"` has become the generalised expression for the direct call of shared libraries.

To get it doing this you must open "**intuition.library**" version 37. That means that you must have at least AmigaOS 2.04 installed. The function "**DisplayBeep()**" is called and the library will be closed at the end.

You have to not three things thereby:

1. Use the right version number because the compiler can not recognise whether you open library version 37 but used a function available from version 38 or later. This will crash a computer running an older operating system.
2. Only call functions from those libraries that you really opened before. Several reasons can prevent opening: There is no library on the computer with the indicated name or it exists one with a smaller version number. Or there is not enough free memory. And there are still many other reasons!
3. Close the library at the latest at the end of the program. For only then the library can be removed from memory again if AmigaOS need it (presupposed no other program uses this library at the same time!).

A further example:

```
#include <pragma/intuition_lib.h>

void main()
{
    DisplayBeep(NULL);
};
```

This program will run too, presupposed you have linked it with "**storm.lib**".

Hmm? - did not we just say that you should open libraries before use. Yes, you should and that is exactly what this program is doing.

Since StormC is a C++ compiler too there are possibilities to automatically call functions from libraries on request at the program start and end. The library "**storm.lib**" contains such functions, which open and close a library on request. This means whenever you are calling a function of a special library it will be open automatically and closed at the end of the program. The linker library "**storm.lib**" knows all OS



libraries up to version 39. Thereby two different strategies are pursued how to treat a failure at opening:

1. Libraries of Kickstart version 37 are opened with just this version and cause an error message and cancellation of the program, if the opening fails.
2. Libraries of later OS versions (e.g. "**locale.library**") are opened with the smallest version (here "**locale.library**" version 38) and the program is continued when the library can not be opened.

You are responsible for calling no functions which are not available on the used OS and you must not call functions from libraries which could not be opened.

The following two code fragment will help you testing this:

```
#include <exec/libraries.h>
#include <graphics/rastport.h>
#include <pragma/graphics_lib.h>
#include <graphics/gfxmacros.h>
external struct Library *GfxBase;

void setoutlinepen(struct RastPort *rp, ULONG pen)
{
    if(GfxBase->lib_Version >= 39)//min. OS 3.0
    {
        SetOutlinePen(rp,pen)
    }
    else
    {
        SetOPen(rp,pen);
    };
};
```

This code distinguishes between AmigaOS 3.0 and older versions to set the pen number for the borders correctly. Older OS version have a macros for this only but as from V39 there exists a function.



The printed listing is not a real program, since the `main()` function is missing. If you want to try this function simply make an own `main()` function for it.

```
#include <libraries/locale.h>
#include <pragma/locale_lib.h>
#include <exec/libraries.h>
external struct Library *LocaleBase;
char yday[80];

STRPTR yesterday()
{ STRPTR retval = NULL;
  if (LocaleBase != NULL)
  {
    struct Locale *locale = OpenLocale(NULL);
    if (locale)
    {
      STRPTR s = GetLocaleStr(locale, YESTERDAYSTR);
      if (s)
      {
        strcpy(yday, s);
        retval = yday;
      };
      CloseLocale(locale);
    };
  };
  if (retval == NULL)
    retval = "Yesterday";
  return retval;
}
```

This function delivers the string **"Yesterday"** of the national language chosen at local settings or **"Yesterday"** if something goes wrong. If you are running AmigaOS 2.04 it is to be foreseen that the program does not deliver the desired result because locale library is provided with AmigaOS 2.1 (V38) first.



Surely it is not skilful to open and close locale structure every time but we want to show the principles.

Forced opening of a Amiga library

If you really want to force opening of the library in a higher version (e.g. **"graphics.library"** version 39) or if you want to open it anyway there exists of course the possibility to open and close the library **"by hand"**. Indeed you should note that you have to declare the basis pointer variable (e.g. **"struct library *GfxBase"**). Otherwise the automatic opening mechanism comes into confusion. You can use the automatic opening mechanism as well and test the **"main()"** function at the beginning and terminate the program when required.



PROGRAMMING OF SHARED LIBRARIES

The creation of an own shared library can be very useful. As soon as several programs use equal code or some functions should be available to other programs or programmers it is meaningful to build a shared library instead of using a collection of functions.

StormC supports you at this task so you must not care about technical details. The creation of a shared library will not be more expenditure than a building a linker library. Nevertheless you should know about the setup of a shared library to avoid some possible errors and to use StormC optimally.

The Setup of a Shared Library

A shared library exists essentially of a basis structure (therefore at least a "**struct library**") and a function table, in which each function of the shared library has an entry. Besides each shared library has a version string, which you can query with the command "**version**" from the CLI.

The basis structure is at least one "**struct library**" but you can add further elements. That is meaningful if the programs, which use the shared library are accessing on some data without having to call a library function.

You should progress to the rule to use public data very sparingly, for the accesses through other programs, whether reading or writing can not be record in the library. You must also pay attention in future versions of the library that these data will keep the same sense, which is not always too simple.

It is safer and more simple for you to hold the data of the library in normal global variables. The access should be by a function, even if these are only very small functions, which deliver the value of a variable. The "**dos.library**" uses this principle intensely since AmigaOS version 36 e.g. to make accessing of structure more save.



If you have divided your library in several modules `#pragma libbase` should be called in one source only.

The `#pragma Libbase`

The name of the basic structure must be told the compiler at a position in the source. This will be done with "`#pragma libbase`".

For example:

```
#include <exec/libraries.h>

struct MyLibBase
{
    struct Library library;
    ULONG second_result;
};
#pragma libbase MyLibBase;
```

Register Setup

The functions of a shared library receive their arguments from a CPU registers, data like "`int`", "`LONG`", "`char`" etc. in data registers ("`d0`" to "`d7`") and pointers in address registers ("`a0`" to "`a5`"). You should count data register starting from "`d0`" and address registers starting from "`a0`". But you should forego "`a5`" since this register is used by almost every C function to set up a so-called "**stack frame**". If a shared library uses "`a5`" as a parameter you will probably get problems when a function is called by "`#pragma amicall`". In this case you should write a stub function for this function in assembler and forego the "`#pragma`" call.

Despite this restraint the address registers "`a0`" to "`a4`" are available and that should suffice for nearly each function.

To be able to access to entries of the basis structure, a pointer to the basis structure in register "`a6`" must be indicated as a parameter. This parameter is set automatically at the call of a function of the shared library.

An example:

```
ULONG add_two(register __d0 ULONG a,
               register __d1 ULONG b,
               register __a6 struct MyLibBase *base)
{
    ULONG retval = a + b;
    base->second_result = retval;
    return retval;
};
```



Shared Library Project Settings

To create a shared library you must create a new project, which contains of at least one source code with some ANSI C functions, a FD file, which describes the register use of the functions and an entry for the program name, which always has the ending ".**library**".

In the settings of the project you choose at the page "**C/C++ options**" "**Large Data Model**" and at "**Linker Options 1**" naturally "**Link As Shared Library**". Besides you can determine the "**version**" and "**revision**" of the library. "**Linker Options 1**" has two fields for this.

The setup of FD files

The FD file looks as follows: their name always uses the suffix ".**fd**", so it is placed in the right section ("**FD files**") of the project. "**FD**" or ".**fd**" stands for "function description").

The file is composed of individual lines, which begin either with a "*" (commentary) or with a "##" (special command). Otherwise the line must contain a valid function description.

Example:

```
##base _MyLibBase
##bias 30
##public add_two(a,b)(d0,d1)
##end
```

The command "**##base**" marks the name of the basis variable, whereby the name is indicated as linker symbol with leading "_". In your C program you will write this variable without the leading "_" character.

The command "**##bias**" sets the offset of the function. In the function table the functions are standing one after another. The first function has the offset -6, the next function the offset -12, the third the offset -18 etc.

With "**##bias**" you can set this offset newly, indeed you will give them positive values. Each function increases the positive offset with 6, for in the function table stands a "**jmp xxx.L**". That is a assembler instruction which jumps at the real beginning of the function. This assembly instruction

requires just 6 byte. Normally you sets this **"##bias"** on 30, for the first 4 functions are reserved.

The command **"##public"** marks the following function as publicly accessible. The command **"##private"** not used in the above-mentioned example marks the following functions as private.

The command **"##end"** terminates the function description. Everything that follows in the file will be ignored.

Before the command **"##end"** stand the function descriptions. These exist respectively of the name of the function, the name of the parameters (which are comma separated encircled by brackets) and finally the list of registers. Every parameter is associated with one register. The parameter of the register a6, which you may have indicated in your ANSI C function, is not indicated here.

The function description increases **"##bias"** automatically at 6. You do not have to set **"##bias"** newly after each function.

Restraints Under StormC there is momentarily a restraint. The command **"##bias"** can be used no more after the first function description. Theoretically it is possible, to create a not coherent function table in the FD file, e.g. because the function table contains functions, which should be invisible to the user. Since the FD file should be used for the creation of a function table it must be coherent.

The first four Functions

Just one word to the reserved first 4 functions. These functions are called **"LibOpen()"**, **"LibClose()"**, **"LibExpunge()"** and **"LibNull()"**. To it comes the function **"LibInit()"**, which is not called over an entry of the function table.

The function **"LibInit()"** will be called and initialised after loading, therefore at the first open of the library. That means particularly at StormC, that the automatic constructors or initialisation functions are called, which will e.g. open other required libraries. That works equal to normal programs, so you must not care for the correct initialisation. For C++ programmer the constructors of the global variable are called too.



The function "**LibOpen()**" is called at each opening and it does nothing than to record just this fact. It increases a counter which indicates how often the library was opened.

The function "**LibClose()**" is called when closing the library and it decreases the counter.

The function "**LibExpunge()**" is always called when the operating system wants to remove the library from memory. That happens for example at memory shortage. The library can be removed only if it was not just opened by a program. The counter must stand on 0, so you will note how important it is to close used libraries. A task of the function "**LibExpunge()**" is to call the automatic destructors or exit functions. Thereby e.g. the automatically opened libraries will be closed again. At use of C++ the destructors of the global variables will be called.

The function "**LibNull()**" does nothing at all and it is reserved for future extensions.

These five functions are in the linker library "**storm.lib**". You can overwrite them at any time with your own functions. Probably you will have to use an assembler for this. A meaningful and also in C feasible expansion would be e.g. to overwrite the function "**LibNull()**" through a function which makes the library useful for ARexx. How this will function you must read in suitable literature.

Home-made Initialisation and Release

Just because you want to execute additional tasks at loading of libraries or when removing them from memory (e.g. the opening of further libraries or memory reservations), that is no reason write the functions "**LibInit()**" and "**LibExpunge()**" new. Simply use the possibility to implement automatic initialisation or exit functions.

If you are using C++ you can do this with global variables, whose type owns a constructor or a destructor. These will be called automatically in "**LibInit()**" or "**LibExpunge()**". The execution sequence can be determine with "**#pragma priority**" more closely.

Using ANSI C you can simply name the functions as the linker expects it to call it automatically. An initialisation function must be named "**INIT_n_anything(void)**" and an

exit function `"EXIT_n_anything(void)"`. Whereby `"n"` is a number between 0 and 9. This number determines the priority of the function. The smaller the value, the earlier the initialisation function and the later exit function is called.

Example:

```
struct Library *DOSBase;
struct Library *intuition base;

INIT_3_OeffneBibliotheken(void)
{
    DOSBase = OpenLibrary("dos.library",37);
    Intuition base =
        OpenLibrary("intuition.library",37);
}

EXIT_3_SchliesseBibliotheken(void)
{
    CloseLibrary(intuition base);
    CloseLibrary(DOSBase);
}
```

Beside these standard functions a shared library still requires certain tables and data structures, which initialise the shared library at start. These are situated at the startup code `"library_startup.o"`, which is used at `"linker as shared library"` automatically. This startup code uses some symbols, which are created by the linker automatically. Among other things a symbol on the beginning of the function table and one symbol on the name of the library and the version string of the library. The two last symbols can overwritten. They are called `"_LibNameString"` and `"_LibVersionString"`.

Important hints to Shared libraries

Here a couple of hints for the development of your own shared libraries.

First develop you functions completely normal and write a test program for them. If everything functions to your satisfaction, transform the project to a shared library, by removing the test program from the project and write a suitable basis structure. Of course some project settings must be changed too.



You have won two advantages by this way: Shared libraries must be removed from memory before you can exchange them by a newer version. Therefore you must care that your test program has closed the shared library and that the old library is removed by executing **"avail flush"** from the CLI. That is much complicated than to simply start the new program. Besides you can not debug in shared libraries.

Pay attention that your functions are "re-entrant". That means that several programs can use it simultaneously. That forbids in general the use of global or static local variables to store interim results. If you want to receive data over several function calls you must work in the certain "object-oriented" way. You know that from **"intuition.library"**: You open a window and get back a **"struct Window *"**. With this pointer you can manipulate the window now, until you close it. Then **"intuition.library"** receives this pointer at each function call and can work with the data structure. Thereby you avoid global variables.

If you need much data locally (e.g. for a large array) you often use static local variables in "normal" programs. In shared libraries you should lay these variables on the stack and if required previously accomplish a **"StackSwap()"** or store the data short-term in the dynamic storage, reserved with **"AllocMem()"**. The third possibility is to protect with a semaphore the simultaneous use of the static variable through several programs. Which method you chooses is a question of the respective situation.



size.

A shared library should not need to use large stack

PORTING FROM SAS/C TO STORMC

We have made it a point to equip the StormC compiler with many important properties of the SAS/C compiler, ie. support for various SAS-specific keywords and **#pragmas**. Nevertheless there may - depending on your programming style - be large or small amounts of trouble when porting software from the SAS/C compiler to the StormC compiler.

Please keep in mind that StormC is an ANSI-C and C++ compiler. SAS/C on the other hand is a C and ANSI-C compiler (the C++ precompiler is not likely to have found much serious use), meaning that it understands a lot of older syntax that StormC will not accept. This is likely to cause trouble when migrating your sources to StormC, unless you are used to compiling your SAS/C programs **strictly in ANSI** mode (using SAS/C's **ANSI** option).

Project settings

First of all make sure that the project you build around your SAS/C sources is in ANSI-C compiler mode.

Try enabling as many warnings as possible, and then adapt your programs until no more warnings are given when compiling. This will give you the best chance that your program will do exactly what you intend it to.

Even for pure ANSI-C projects, switching to C++ later is recommendable. This will have several advantages: Prototypes are expected for all functions, and implicitly converting a **void *** to another pointer type is no longer legal.

Although this may necessitate a relatively tiresome rework of your programs (especially the latter change which affects a great deal of statements that call **malloc()** or **AllocMem()**), but can give you a great deal more confidence in the correctness of the program.

The long symbol names in C++ provide additional security while linking: If a function definition is in any way inconsistent with its prototype declaration, the linker will abort with reports of an undefined symbol.

Switching to C++ will also give you the possibility to extend your program with modern object-oriented concepts, as well



as the use of several smaller C++ features (such as the ability to declare variables anywhere in a statement block).

Syntax

Some SAS/C keywords are not recognized by StormC, others are supported well, but the more "picky" StormC compiler only allows them in the typical ANSI-C syntax.

StormC does accept anonymous **unions**, but not implicit structs. Equivalent structures are not considered identical. If you have made use of this feature, you will need to insert casts in some places.

If this feature is important to you, you may want to consider moving your project over to C++: Equivalent structs are nothing but (an aspect of) inheritance in a different guise.

Type matching is much more strict in StormC. This is especially the case for the **const** qualifier used on function parameters. An example:

```
typedef int (*ftype)(const int *);
int f(int *);
ftype p = f; // Error
```

For such errors you should either insert the necessary casts, or (and this is always preferable) write the appropriate declarations for your functions. After all the **const** qualifier is an important aid in assuring the correctness of your program.

Rest-of-line comments as in C++ ("//") are accepted even in ANSI-C mode, but nested C-style comments are not. In any case you can enable the warning option that detects these dangerous cases.

Accents in variable names are not accepted, nor is the dollar sign.

Keywords

The use of non-standard keywords is generally best avoided - at least for programs that you may want to port to another operating system or a completely different compiler someday.

StormC makes more use of the `#pragma` directives provided by ANSI-C for adapting software to the special requirements of AmigaOS (eg. `#pragma chip` and `#pragma fast`).

For keywords that may not exist in other compiler systems but are not absolutely necessary, the use of special macros is recommended:

```
#define INLINE __inline
#define REG(x) register __##x
#define CHIP __chip
```

These macros can then be easily modified to suit a different compiler environment.

Some optional keywords not recognized by StormC can also be defined as macros:

```
#define __asm
#define __stdarg
```

Here's a list of SAS/C keywords and how StormC interprets them:

`__aligned`

is not supported. There is no simple way to replace this keyword, but fortunately it is rarely needed.

`__chip`

This keyword forces a data item into the ChipMem hunk of the object file. Note that this keyword, like all other memory-class specifiers and other qualifiers must precede the type in the declaration:

```
__chip UWORD NormalImage[] = { 0x0000, .... }; //
correct
UWORD __chip NormalImage[] = { 0x0000, .... }; //
error
```

The latter syntax is not accepted as it is not consistent with ANSI-C syntax.

In StormC the use of `"#pragma chip"` and `"#pragma fast"` is preferred. Take notice however of the fact that `"__chip"` affects only a single declaration whereas `"#pragma chip"` remains in effect until a `"#pragma fast"` is found.

`__far and __near`

are not supported. There is no easy way to replace these keywords, but they are rarely needed.



is not supported. At the moment all interrupt functions (a rarely needed class of functions anyway) must be written in assembler.

`__interrupt`

are not supported and not needed. If you wish to have function arguments passed in registers, declare the function with the ANSI keyword **"register"** or modify the individual parameter declarations with the **"register"** keyword or a precise register specification (eg. **"register __a0"**). Otherwise the arguments will be passed on the stack.

**`__asm`, `__regargs`,
`__stdarg`**

Has an effect similar to SAS/C's **"__saveds"**. This keyword has no effect when using the large data model; in the small data model relative to **a4** it saves **a4** on the stack and loads it with the symbol **"__LinkerDB"**, in the small data model relative to **a6** it does the same for **a6**.

`__saveds`

Do not use **"__saveds"** lightly. It should be used exclusively for functions that will be called from outside your program, eg. Dispatcher functions of BOOPSI classes.

In the current compiler version it is recommended to use only the large data model when creating shared libraries. Remember that the small data model makes yet another register unavailable to the optimizer leaving only **a0** to **a3** - this can quickly nullify the advantage of using the small data model if you're not using a great deal of global variables.

Like the others, this keyword is accepted as a function specifier.

`__inline`

This means that their usage in a function definition must match the prototype.

If an **"__inline"** function is to be called from several modules, its definition (not just its prototype) should be placed in a header file.

is not supported. Stack checking or automatic stack extension is not available at this time.

`__stackext`

CLI VERSION OF THE COMPILER

There should actually be people who do not like the integrated development environment. Therefore StormC can be used as CLI version too. You will only need the program StormC from the drawer "**StormC:StormSYS**".

The concept The compiler creates object files from the sources, which will be bound to a program by StormLink. Without special option the compiler translates all indicated sources and writes the object files.

The instruction

StormSYS/StormC main.c text.c unter.c

compiles the three sources "**main.c**", "**text.c**" as well as "**unter.c**" and creates their object files. Subsequently you must link the object files with the linker to get a finished program.

If an error appears at the translation a suitable output with a short portion of the source is output, along with an output of the kind

**File "filename.c", line 17:
Error: Identifier "ths" not defined.
Go on with <Return>, abort with <A>:**

Now you can pass over the error with <Return> and continue the translation or press <A> to abort the compilation.

StormC is capable of being resident, even if some Shells announce a sum error. This is harmless in each case, even if StormC was made re-entrant on a somewhat unusual method: Using a semaphore a loaded copy of the program cares for that it does not run twice simultaneously.

Options

All compiler functions are steered through compiler options, which are indicated in the command line and begin respectively with a minus character "-".

To bring a little order in this extensive jumble, the options can be divided using their respectively first letter into different ranges:



- **a** Assembler Source Generation
- **b** Debugs
- **d** Symbol Definitions
- **e** Error Messages
- **g** Code Generation
- **h** Pre-Compiled Headers
- **i** Paths for Includes
- **j** Paths for Libraries
- **l** Linker Libraries
- **p** Parser
- **s** Copyrights
- **w** Warnings and Main Memory Size

Options are always global: At a line of the form

StormSYS/StormC -x text1.c -y text2.c -z

"**text1.c**" and "**text2.c**" will be compiled equally with the options "**x**", "**y**" and "**z**". In other words: The sequence of file names and options is not relevant.

The sequence of the options mutually is not alike, if contradictory options appear in an instruction line. For instance you turn on with "**-gO**" the optimisation and with "**-go**" off. In cases like

StormSYS/StormC -go -gO etc.

the respectively last statement is valid.

Not only for the optimisation, but for almost each feature there are two options: one to switch on and one to switch off. At first glance that appears superfluously, because e.g. the optimisation is not activated by default and so an option for switching it off is not really necessary.

At the most switching possibilities the way was chosen, that you switch on with an uppercase letter and off with a lowercase letter like "**-go**" and "**-gO**".

The most important example is the option "**-b**" for the creation of debugger files. The option switches are defined through appendices of a Zero. "**-b0**" means therefore "**No debugger files**".

In some cases abbreviations are introduced also, e.g. "**-o**" is identically with "**-gO**". A two-letter-combination through

an individual letter to abbreviate, is perhaps no particularly impressive saving. In this particular case **"-o"** has the advantage to be compatible to other compilers.

Assembler source

-a and -as

Perhaps you would like to examine once the code created by StormC, or you belong to the people, who want to optimise the compiled program by hand. With the option **"-a"** you can get the output of an assembler listing.

E.g.

StormSYS/StormC test.c -a

creates not only the object file **"test.o"** but also the assembler listing **"test.s"**. The statement **"-a"** does not abort the translation process before the assembling, but creates the source in addition to the remaining actions.

A pure assembler listing is rather tangled in most cases. Therefore there is the alternative **"-as"**, through the statements from the original sources will be inserted at the respectively corresponding positions as commentary in the assembler source.

To switch off these both options explicit you must use **"-a0"**.

Pre-processor: Definition of symbols

-d

With the option **"-d"** you can define "from outside" a pre-processor symbol. Thereby an argument corresponds to

-d NAME

a line

#define NAME

at the beginning of the source. A typical application is an instruction like

StormSYS/StormC test.c -d NDEBUG

if the include file **"<assert.h>"** is used.



You can assign a value to the symbol by setting "=" directly behind the identifier, e.g.

-d ANZAHL=4711

As value a token is allowed here only, that means at

-d N=17+4

would ignore that "+4".

Pre-processor: Include files

The path where StormC searches the include files is by **-i** default set to "**StormC:Include**".

Through one or several "**-i**"-options you can change this path: With the option

-i StormC:inc

the drawer "**StormC:inc**" will be looked for exclusively, and at

-i StormC:include -i StormC:include/amiga

the pre-processor searches successively in the both drawers.

Compiler mode

ANSI C or C++

StormC can also be used as ANSI C compiler. If you do not want to set "**#pragma -**" at the beginning of the source, you can use the compiler option "**-pc**". C++ is the pre-set mode, which is the same as the option "**-pp**". Therefore "**c**" like "**c**" and "**p**" like "**plus plus**" - this is however completely easy to keep in mind.

-pc and -pp

Exception handling

Sometimes it is advantages, if you translate a "classic" C++ program, which does not use exception handling, in a particular compiler mode. Then the key-words ("**catch**", "**throw**" and "**try**") connected with the exception treatment are recognised no more, and in the created code there is no bookkeeping of the necessary destructor calls. That will make C++ programs faster and smaller. ANSI C programs will not be affected.

-px, -pX and -X

As default the exception handling is switched off ("**-px**"). If you would like to use the exception treatment you must turn it on first with "**-pX**". As abbreviation "**-X**" is allowed too.

Creation of Template functions

-t and -T

It is not always easy for the compiler to decide, which Template functions it should create and which not. "**-T**" corresponds to the strategy "**All Templates Create**", at "**-t**" is pursued the other strategy to create what will be created faultlessly.

Code creation

Data model

-gd, -gd4, -gd6 and -gD

The option "**-gd**" switches to "**NearData (a4)**" and "**-gD**" to "**FarData**". Pre-set is "**FarData**". With the options "**-gd4**" and "**-gd6**" you can distinguish additionally, whether the data model should be created relatively to the address A4 or to A6. The last one is very interesting at the creation of Shared Libraries.

Code model

-gc and -gC

The option "**-gc**" switches to "**NearCode**" model and "**-gC**" to "**FarCode**" (default).

Optimisations

-go, -gO and -O

The additional optimisations are turned on with the option "**-go**" and off with "**-gO**". Alternatively you can use for "**-go**" also "**-OO**". By default they are off. "**-O**" can be used as abbreviation for "**-go**". If the optimiser is turned on only, the compiler optimises in the supreme optimisation level.

Instead of switching the optimiser on and off there is the possibility to switch to certain optimisation levels.

-OO

With the option "**-OO**" the optimiser will be switched off. This is the same as "**-go**".

-O1

This switches the optimiser to the first level where basic blocks are optimised. A basic block is a sequence of intermediate code instructions, which contain no jumps. Successive blocks are summed up, which makes work easier for later optimisation steps. Unused blocks are removed completely. The step is repeated until nothing more is to be improved on this level. Also after higher optimisation steps



it is tried to sum up basic blocks and to remove unused blocks.

Useless instructions like e.g. assignments at variables, which are never used again will be removed. **-02**

Variable and interim results are packed into processor registers. **-03**

Assignments at variables will be eliminated if the variable is never used again. If a complete instruction can be eliminated that way the whole intermediate code will be optimised again, until no superfluous assignment is found any more. **-04**

Example:

In this complete useless function

```
void f( int i)
{
  int j = i+1;
  int k = 2*j;
}
```

only the second instruction will be recognised as useless at lower optimisation levels. From level 4 on the code is perused subsequently once more and then the first instruction will be eliminated too.

At the M680x0 code generation MOVE commands to destination register, which are used subsequently only once before they receive a new value, are pushed together if possible. So **-05**

```
move.l    8(a0),d2
add.l     d2,_xyz
```

becomes thereby

```
add.l     8(a0),_xyz
```

At the interpretation of expressions the interim results of all kind will be laid to auxiliary variables at code generation and packed to processor registers later. At lower level these auxiliary variables will be recycled if possible to keep their number low. The expression "**a*b+c**" would create the following intermediate code instruction: **-06**

```
h1 = a*b
h1 = h1+c
```

From level 6 on the auxiliary variables are never used again. The generated intermediate code would look like:

```
h1 = a*b
h2 = h1+c
```

In later optimisation step there will be a check whether it makes sense to pack "**h1**" and "**h2**" in the same registers. That costs time and RAM because of the increasing number of internal auxiliary variables, but it allows to make a good distribution of the registers.

Code for special processor

-g00- -g60, -g80 and -g81

Also there are always fewer Amiga-User which use a normal 68000 processor StormC normally generates code for this processor. It is compatible to all further processors of this family. Indeed Motorola implemented some new commands to newer processors beginning with the 68020. They will accelerate programs partially, i.e. at multiplication and at array access. The following options switch to the code generation for certain processors:

```
-g00 68000    (default)
-g10 68010
-g20 68020
-g30 68030
-g40 68040
-g60 68060
```

In the current version of StormC the options "**-g20**" and "**-g30**" are identical. Indeed the options "**-g40**" and "**-g60**" differ drastically from the others. Since some FPU functions are not implemented, they must be emulated. At code generation for 68040/060 processors particular libraries are used to create no FPU commands like it is done in other compiler systems.

The use of a Floating Point Unit (FPU) will speed up programs that use many floating point operations ("float" and "double"). With the following options StormC creates code for FPUs:

```
-g80 without FPU (Default)
-g81 68881
```

-g82 68882 (identically with **-g81**)

Code for linker libraries

If desired, StormC can put each function and each global variable in separate hunks. As the linker does not takes non-referenced hunks into the executable program file, this mode suits particularly for linkable libraries. With **"-gL"** or **"-L"** this option is turned on. **"-g1"** switches to normal mode. **-gl, -gL and -L**

Debugger

RunShell

"-b" creates the necessary debugger files (**".debug"** and **".link"**). **"-b0"** switches to original mode. **-b, and -b0**

Symbolic debugger

The option **"-bs"** creates symbol tables hunks for symbolic debuggers. **-bs and -bs0**

Copyrights

The option **"-s"** outputs the following text at the call of the compiler: **-s**

**Programname versionnumber (creationdate),
copyright, author**

Example:

StormC 1.1 (07th05.96), Copyright (c) 1996 HAAGE & PARTNER computers Ltd.

Warnings and errors

Format of the error output

StormC can output error messages in three different kinds: interactive, in short or long format. The pre-set is "interactive" error output, for which also the option **"-ei"** can be used. Thereby for each error message a short source section (approx. 5 lines), the source position and a prompt (**"Go on with <return>, abort with <A>:"**) is shown. The exact error position is emphasised in the source section colourfully. After pressing <Return> or alternatively <Space> or <C> the translation is continued. At input of <A> or alternatively <ESC> or <Ctrl>+<C> the translation is aborted. **-e, -el and -ei**

The compiler option **"-e"** switches on short output. Here only two lines, the source position and a short message are outputted at each error and each warning. After ten errors the translation process is aborted anyway. You can choose another value for that number if you append it just after the option i.e. **"-e20"**.

The long format is a mixture from interactive mode and the short one. Thereby sources and error message are outputted like at **"-ei"** but it does not wait for an input. After a maximum of ten errors the compiler aborts the action. For this form of the error output you choose the option **"-e1"**. As on the option **"-e"** you can choose the abortion level by adding a number. E.g. **"-e15"** would be a meaningful setting, since ten messages would not fit into a normal Shell window.

Colours and styles

-eb, -eB and -ec

The however somewhat boring text output can be made more clearly. There is the possibility to show the real error messages in bold with the switch **"-eb"**. **"-eb"** switches back to basic attitude. The option **"-ecN"** brings colour in the game, at least if you use a numeral for **"N"**. The screen colour number **"N"** is used to differ between the source section of other outputs. **"-ec1"** corresponds to the standard text colour and **"-ec0"** is (theoretically) the background colour. Since you could not read the text any more in the more last case, so **"-ec0"** outputs the source in colour 2, but the error position in an other colour than **"-ec2"**.

Error file

-ef

For the linking of StormC in foreign environments it can be useful, to reroute all error messages in a file. That is done by a statement like **"-ef <name>"**. Thereby all error messages and source positions are written in the indicated file. The difference to a simple output routine is that **"-ef"** works in addition to the normal screen output and secondly it does not appear as a status messages in the file.



Optional warnings

You turn on a optional warning by placing their character as **-w** BIG ones behind that **"-w"**, while a small letter switches it off.

These are the warnings and their characters:

- A** (Assignment): The operator **"="** appears in a condition - perhaps a the will-known **"=="**-novice-error? You can inhibit such a warning by an additional brackets, i.e. **"if ((a=b))"** instead of **"if (a=b)"**.
- C** (Conversion): Uncertain type conversion without cast, i.e. double -> int.
- E** (Empty statement): Instruction has no code, i.e. **"36;"**
- F** (Function prototypes): Function called without a prototype (only meaningful in C as in C++ it is an error).
- N** (Nested comments): A **"/*"** emerges in a comment.
- O** (Old-Style): Complaining about parameter lists in the old **"K&R"**-style (only meaningful in C as in C++ this is an error).
- P** (Pragma): Unknown **"#pragma"**.
- R** (Return): Function with return value has none **"return"**-statement.
- T** (Temporary): Temporary object was introduced, to initialise a non-const-reference, which is an error in C++ since standard 2.0.
- V** (variable): Local variable is used never, or used, but never initialised.
- M** (Macro): An argument for a pre-processor macros extends over more than 8 lines or 200 tokens. It is obvious that there is a closing bracket missing.

By default **"-wEPRT"** is activated. I.e. to switch warnings at missing prototypes on and off such at unknown **"#pragma"**, you can simply use the option **"-wFp"** or alternatively both options **"-wF -wp"**.



Please note, that `"-w+v"` does NOT mean that the warning `"v"` is turned on in addition to the current, but that completely all warnings are switched on and subsequently `"v"` off again.

To turn on all warnings - by the way a sign for good programming style - is some typing work. To make it a little easier there is an abbreviation: a `"+"` directly behind that `"-w"` turns on all warnings.

You can also switch off all warnings with `"-w-"`, but I would not advice this to you. A "clean" program should output any warnings, at most `"a"`-warning are possibly as printout of your personal programming style. If you take warnings actually so seriously, the following section will be very interesting for you.

Treat warnings like errors

`-ew and -eW`

With the option `"-ew"` warnings are treated like error messages. `"-ew"` switches back to normal mode.

Core memories

`-w`

Of course StormC allocates the memory for the most internal processes automatically and dynamically, but at code generation there must be a coherent memory area. This size is 80 KByte by default. If there is the message "Workspace overflow!" you should increase the amount with the option `"-w"` cautiously, i.e. `"-w150"` for 150 Kbytes.

Pre-compiled header files

`-H, -h and -h0`

StormC can pre-compile parts of a program. That means that the compiler can store all its internal data like declarations and definitions, at a user selectable position. At the next run of the compiler this file can be loaded very quickly and so it saves much time.

This feature is for speeding up program development, especially when many header files are used i.e. the AmigaOS-Includes. To tell the compiler where the headers ends and your program begins, you must mark this position with the command `"#pragma header"`.

It is best to write the `"#include"`'s of the header files you did not do by your own or which you will not change any more at the first place in the source of all units. Then the line `"#pragma header"` follows. Next parts are the include instructions for your own header files and the rest of your program.



Without special option or with the explicit switch "**-h0**", "**#pragma header**" has no effect. If you indicate an option "**-H <name>**", the compiler stops shortly as soon as it reaches the pragma and writes all definitions and declarations into the file "**<name>**". To accelerate the program compilation now you must choose the option "**-h <name>**". Then the compiler reads the pre-compiled headers from the indicated file, searches the source for "**#pragma header**" and continues this work from there.

Since everything which stands before "**#pragma header**" is completely ignored, it has to stand consequently in the main source file and not in an include file.

Please keep in mind that the compiler does not check, whether the content of the pre-compiled header files still corresponds with the original source. Therefore I recommend to compile such header files only which will not be changed any more.

Summary

Now a short, alphabetically organised summary of all options follows:

- a** Creates an assembler source for each translated text file.
- as** C source is inserted as commentary into assembler source.
- a0** No assembler output (default)
- b** Create small debug information file.
- bf** Create large debug information file.
- b0** Without debug information file (default).
- bs** Create "symbol hunk" for symbolic debugger.
- bs0** No symbol hunk (default)
- c** Compile sources and creates the corresponding ".o" files.
- d NAME** Defines a pre-processor symbol. Optionally a "=" and the desired value (only a token) can be set directly behind the name. Otherwise the symbol is defined as "**empty**".
- eB** Error messages written in bold.
- eb** Error messages written normally (default)
- ecN** Output source in colour number "**N**" at error messages.
- ef xxx** Write error messages to file "**xxx**".

-ei	Ask the user whether he wants to abort at each error message during compilation (default).
-e[N]	Output error messages without user interaction. After " N " errors compilation is aborted (default is 10).
-el[N]	Like " -e ", but with source output in addition.
-ew	Treat warnings like errors.
-eW	Output warnings, but ignore them (default).
-gd	Data model " Near Data (a4) ".
-gd4	Same as " -gd ".
-gd6	Data model " Near Data (a6) ".
-gD	Data model " Far Data " (default).
-gc	Code model " Near Code ".
-gC	Code model " Far Code " (default).
-gl	All functions in one section (default).
-gL	Library mode: Create an own section for every functions and global variables.
-go	No optimisation (default).
-gO	With optimisations.
-g00	Generate code for 68000 (default).
-g20	Generate code for 68020.
-g30	Generate code for 68030.
-g40	Generate code for 68040.
-g60	Generate code for 68060.
-g80	Do not generate code for FPU use (default).
-g81	Generate code for 68881-FPU.
-g82	Generate code for 68882-FPU.
-H xxx	Store pre-compiled header files.
-h xxx	Load pre-compiled header files.
-h0	Do not use pre-compiled headers (default).
-i xxx	Set path(s) StormC should use to look for includes. It is possible to use multiple " -i " for adding more paths. The default is " StormC: include ".
-L	Abbreviation for " -gL "
-o xxx	Sets an other name for the object file.
-O	Abbreviation for " -go "
-00	Alternative for " -go "
-01	Turns on the optimiser level 1.
-02	Turns on the optimiser level 2.
-03	Turns on the optimiser level 3.
-04	Turns on the optimiser level 4.
-05	Turns on the optimiser level 5.
-06	Turns on the optimiser level 6.
-pc	Switches to ANSI C mode.
-pp	Switches to C++ mode (default).
-px	No exception handling.

- pX** With exception handling.
- s** Outputs the copyright message and the version number of StormC.
- t** Create faultless template functions only.
- T** Create all required template functions.
- w xxx** Output warnings. An arbitrary sequence of characters can follow. Big ones will turn on warnings, small ones will turn them off.

There are the following warnings:

- A** (Assignment): Operators "=" appears in condition. Perhaps the will-known "=="-novice error?
- C** (Conversion): Uncertain type conversion without cast, i.e. double -> int.
- E** (Empty statement): Instruction has no code, i.e. "44;"
- F** (Function prototypes): Function called without a prototype (only acceptable in C, because in C++ it is an error)
- N** (Nested comments): "/*" in commentary.
- M** (Macro): An argument for a pre-processor macros extends over more than 8 lines or 200 tokens. It is obvious that there is a missing closing bracket.
- O** (Old-Style): parameter lists in the old "K&R" style (only acceptable in C, because in C++ it is an error).
- P** (Pragma): Unknown "#pragma".
- R** (Return): Function with return value has no "return".
- T** (Temporary): Temporary object was introduced to initialise a non-"const" reference (this is an error since that 2.0 standard).
- V** (variable): Local variable is never used and/or used, but never initialised.

By default "-wEPRT" is activated.

- wN** Sets the interim compiler workspace on "N" Kbytes. Default is 80 Kbytes, which should be enough for most cases.
- w+** Output all warnings.
- w** Output no warnings.
- X** Abbreviation for "-pX".





After the first successful compilation of a program the test phase starts. Of course you could simply run the program from a CLI (or a Shell), yet this can cause some typical Amiga problems:

1. If the program makes serious errors, which is always possible in C or C++, the computer may crash, because the CPU has so called exceptions, i.e. special soft- or hardware states. These are treated only very generally by the OS, and often don't allow you to work on in a reasonable manner without rebooting the system.
2. Rebooting the Amiga is the only solution in most cases when the program is running into and endless-loop.
3. After an exception or when the program is stopped by "**exit()**" or "**abort()**" most of the system resource are still reserved. While still reserved memory is only annoying, open files or windows may hinder the further work very much or even lead to crash of the OS.
4. The problems mentioned here cannot be traced back to their origin in the source code.

GENERAL INFORMATION ON RUNSHELL	151
The StormC Maxim	151
A Resource-Tracking Example	152
Freeze the program temporarily	153
Halting the program	154
Changing the task priority	154
Sending signals	154
USING THE DEBUGGER	157
The Variable Window	159
Temporary Casts	161
Changing Values	161
Sorting of Variables	161

THE PROFILER	164
Profiler technical information	167
REFERENCE	169
Control Window	169
<i>Status line</i>	<i>169</i>
<i>"Program Stops At Breakpoint":</i>	<i>169</i>
<i>"Continue Program":</i>	<i>169</i>
<i>"Program waits for ...":</i>	<i>169</i>
<i>Debugger icons</i>	<i>169</i>
<i>Go to next breakpoint</i>	<i>170</i>
<i>Step in (single step)</i>	<i>170</i>
<i>Step over (single step, but execute function calls without stopping)</i>	<i>171</i>
<i>Go to the end of the function.</i>	<i>171</i>
<i>Show Current Program Position</i>	<i>171</i>
<i>Pause</i>	<i>172</i>
<i>Kill</i>	<i>172</i>
<i>Priority gadgets</i>	<i>172</i>
<i>Signal group</i>	<i>172</i>
<i>Protocol gadgets</i>	<i>172</i>
<i>Window close gadget</i>	<i>173</i>
Current variable window	174
The module window	177
The function window	178
The history window	178
The breakpoint window	179
The address requester	180
The hex editor	181
<i>Choosing the display</i>	<i>181</i>
<i>The address string gadget</i>	<i>181</i>
<i>The address column</i>	<i>181</i>
<i>The hexadecimal column</i>	<i>181</i>
<i>The ASCII column</i>	<i>181</i>
<i>Keyboard control</i>	<i>182</i>
<i>The scrollbar in the hex editor</i>	<i>182</i>



GENERAL INFORMATION ON RUNSHELL

It is for the four reasons of the introduction to this chapter StormC has a "RunShell". It will launch the program similar to a start from the CLI, yet it caters for a whole range of ways to control and direct the program.

A special feature is "Resource-Tracking", which solves problem 3, for example. After the program is interrupted or finished, all resources that have been allocated but not freed will be shown. The resources are then freed by the RunShell. Next the position in the source code where the resource was allocated can easily be jumped to.

Connected to this RunShell is the Source Level Debugger. It allows you to execute programs step by step, localise errors, or inspect variables. But also directly control the program, and always find the origin of problems (in the worst case a crash) in the source code.

Additionally it is possible to halt the program (for example in endless loops), without having to compile the program in a special way.

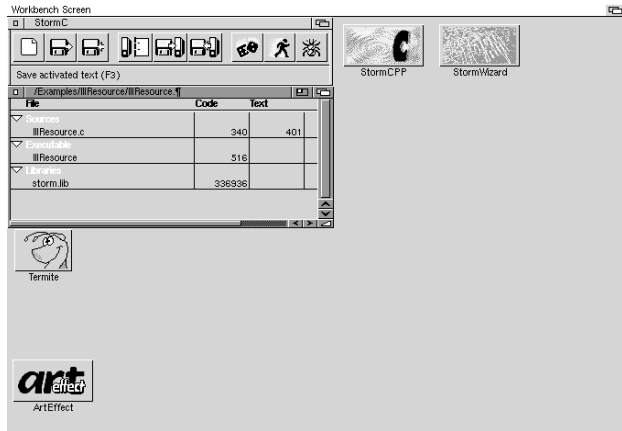
The StormC Maxim

The program is in the test phase is equal to the final product. To be able to use the source level debugger you need to compile with the corresponding option, yet this doesn't influence the generated program. This prevents the problem other systems from time to time have: bugs that show up when running a program under normal conditions can't be reproduced using the debugger.

A further special feature is the ability to switch from normal execution to the debugger, and to return from the debugger to normal execution (if the program was compiled using the "debug" option).

A Resource-Tracking Example

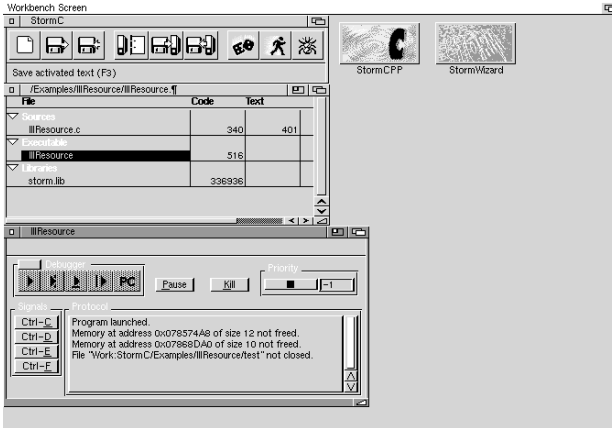
Open the project "**IllResource.1**" in the "**StormC: Examples/IllResource**" drawer. This program allocates some resources without freeing them.



Start the program using for example the function key F9. Subsequently the program will then be compiled if necessary and then run.

Other ways of launching your program are: selecting the button in the toolbar with the walking man on it, selecting the menu item "**Start**" from the "**Compile**" menu, double clicking the entry for the program in the project window, or directly after compiling by hitting the "**Start**" button in the error report window.

After compiling the Runshell control window will be opened. This is your "control board" for the program.



The status-line shows help texts for elements of the control window and also the status of your program, when you're not above any control gadgets.

The "**IllResource**" program now shows that it's waiting (the "**wait()**" function of "**exec.library**"), and also shows the mask of the signal flags that it is waiting for. At the moment this reads "**0x00001000**", which corresponds to a Ctrl-C.

The icons in the "**Debugger**" group aren't needed for the moment, first we want the program to execute normally. The debugger can be activated with the checkmark on the group frame, the icons in this group stand for "**Go**", "**Step in**" (single step), "**Step over**" (single step, but execute function calls without stopping), "**Go to end of function**" and "**Show last found program position**". These functions will be explained later in the debugger introduction.

Freeze the program temporarily

The "**Pause**" button allows you to freeze the program at any time, e.g. when you wish to inspect the contents of a window, that would have been overwritten quickly otherwise. This button toggles, first the button is "**lowered**", and stays that way. The program will be frozen until you "**raise**" the button again by selecting it once more.

The program will continue to execute. The menu item "Pause" in the "Debugger" menu also freezes the program.

Halting the program

Next to it is the "**Kill**" button. This martial action allows you to end program execution at any time. The menu item "**Kill**" in the "**Debugger**" menu has the same result.



Epecially by setting too high a priority for your program you can block the system. Please be careful when changing this setting.

Changing the task priority

The "**Priority**" group shows the priority of the program and allows you to change it. Even though the priority can be changed throughout the whole range of -128 to 127, it should be restricted to the range of -20 to 20, otherwise troubles with other programs or even an unusable system can result.

The value set at program start (normally "1") is really a sensible one, because it is one step minor than the priority of the debugger, so working with the debugger is always fluently.

Sending signals

The "**Signals**" group contains four buttons, to send the four interruption signals of AmigaDOS:

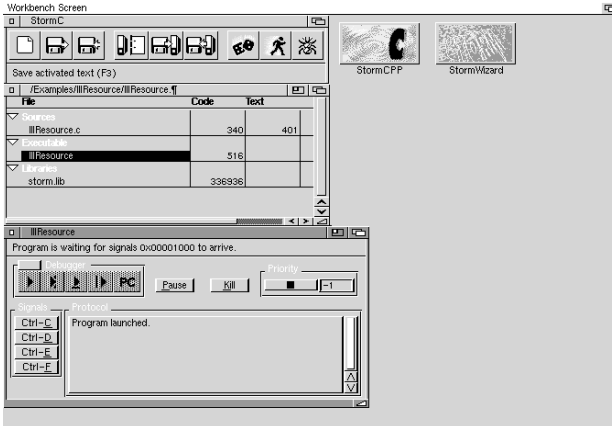
Ctrl-C, Ctrl-D, Ctrl-E, Ctrl-F.

You can very easily control a program while testing by using corresponding "**wait()**" statements. Much like the program "**IllResource**" that currently waits for a Ctrl-C before it allocates the resources.

Now select the "*Ctrl-C*" button in the "**Signals**" group.



The program has now allocated some resources, but it didn't free them again. The program has finished.

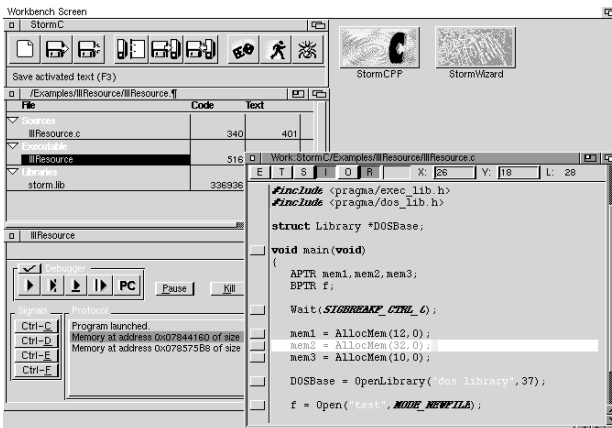


The protocol list shows information about resources that were allocated and not freed. This involves memory allocations, libraries, and a file.

Double-click a line in the protocol list, and the position in the source code where the resource was allocated will be shown. If the source code wasn't available in an editor to begin with, a new window will be opened and the source code will be read.



Even though Resource-Tracking always works and the Runshell always frees the resources, the source code position can only be shown if the program was compiled with debugging information.

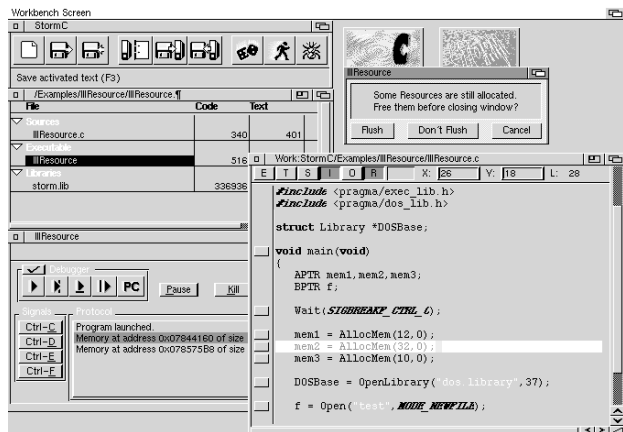


In a real program you could now start on writing the appropriate code to free the resources. Where this should be done of course depends on your program.

To be able to show the position in the source code where resources are allocated incorrectly, one condition must be fulfilled:

The corresponding OS function must be called directly in the source code. This rules out calls from link-libraries, i.e. OS functions should not be called via **"amiga.lib"** and the stub-functions contained therein, but instead always use the correct **"#pragma amicall"** and **"#pragma tagcall"** calls.

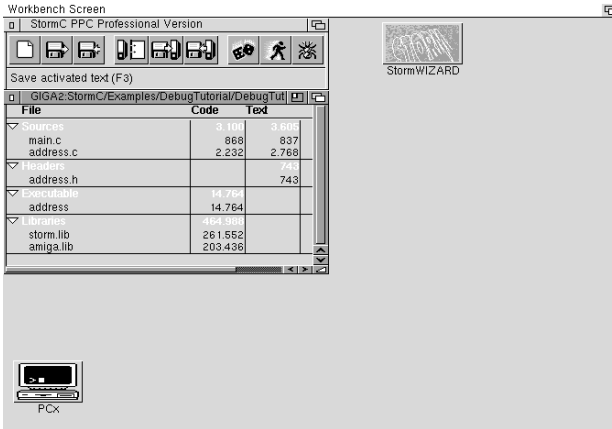
Now free the resources using the menu item **"Free Resources"** from the **"Debugger"** menu, or directly close the control window, which will automatically close the resources.





USING THE DEBUGGER

Load the project "**StormC:Examples/Debugger Tutorial/DebuggerTutorial.1**". The program consists of many modules with one header file, just enough to also show some non trivial debugging situations.



Start the program normally and get familiar with it. First a menu with three items will appear:

- enter an address,
- show all stored addresses and
- end of program.

If you wish to enter a new address, enter 1 (don't forget *<Return>*). Then enter a person's last name, first name, street and place of residence. Repeat this two or three times and have a look at the result by choosing menu item 2. Then end the program.

Because normally you only debug your own programs, you should now have a look at the source to get familiar with the function names.

The main program is in "**main.c**". It shows the main menu and evaluates the menu input. Depending on the input functions from "**address.c**" will be called.

First up in the **"address.c"** module are some static functions for use by the other functions. Then all the address handling functions follow.

The header file **"address.h"** contains the **"struct Address"** data structure and the prototypes of all functions that do something with addresses, particularly input and output. These functions are implemented in the **"address.c"** module and are called by the **"main.c"** module.

When so far everything is clear, start the program again, only now in the debugger, i.e. with *F10* or the corresponding icon in the toolbar.

Other ways to debug a program are: the menu item **"Debug"** in the **"Compile"** menu, double clicking the program name in the project window while simultaneously holding the <Alt> key, or directly after compiling in the error window by selecting the **"Debug"** button.

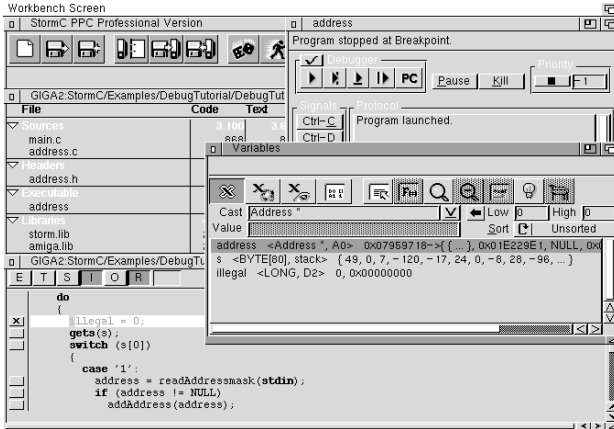
Again the control window is opened first, only now with activated debugger controls. The program will automatically execute to the first position in the program for which source code is available. If you haven't already loaded the **"main.c"** source, it will happen now. The editor window now looks a bit different: a column of breakpoint buttons is located to the left of the text.



A breakpoint button is there for every position in the source where the program can be stopped, breakpoints are the positions in the program where execution will stop, when it was started with "Go" (and runs at full CPU speed).

In this column you can set or clear a breakpoint with the mouse. This column also shows how far a single step takes you, since often a step in C takes you more than one line

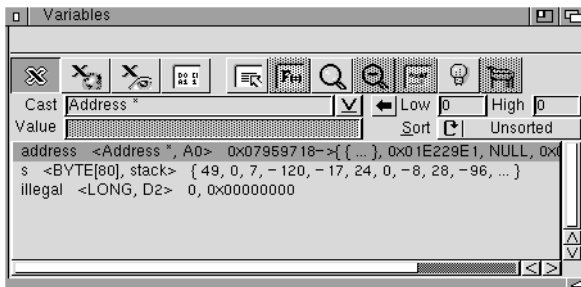
ahead (for example skipping comments, declarations, long expressions etc.).



The current program position is always in the editor shown using a white bar that marks the whole line horizontally.

The Variable Window

Subsequently the window with the variables is opened.. This always shows all variables that are available to the program at this time.



At the first line of the window a short help text is shown, which tells you the meaning of the different buttons of this window.

Below this line is the button line. On the left side there are 4 buttons to select one of the 4 variable pages.



The 1st page shows all parameters and local variables of the current function and the global variables of the module that contains this function.

The 2nd page contains all global variable of all modules.

The 3rd page shows the variables that should be supervised. This can be local variables of functions or even parts of a structure (members).

The 4th page contains the cpu register values.

Choose the first variable ("**address**") out of the list of the current variables, by selecting its name. Next you can also reach other elements of the window.

Next to the buttons for paging you will find these for actions on the selected variables.



By selecting the 1st button the source code position where the variable is defined will be shown by positioning the cursor there.



The 2nd button is interesting for C++ programmers. It shows a list of all member functions of the type of this variable. This list will be empty, because in this example there are no member functions.



The 3rd button inspects a variable. The variable will be looked for very exactly e.g. every field of a structure is shown individually.



The 4th button returns from the inspection and the value of the variable list will be the same as before the inspection.



The 5th button opens the hex-editor and puts the cursor at the address of the variable.



The 6th button adds the variable to the watched variables window. This allows you to build a list of variables that you always want to see, even if the program is not executing the corresponding module or function.



The 7th button is only available on the page of the watched variables. It deletes a variable out of the list of watched variables.



Temporary Casts

The string gadget "**Cast**" allows for easy casting of the type of a variable. You may enter a type here that is defined somewhere in the module. Pay attention to write the type like it is shown in the variable list. If the type is found the value of the variable will be interpreted according to the new type. E.g. it might be sensible to make an "**ULONG**" a "**LONG**" to get the signed value. The most often used application is to change the target type of a pointer, e.g. to make "**Message ***" an "**IntuiMessage ***". At next inspection of the variable the new type will be used.

The old type will always be shown in the variable list. The new one is only in the string gadget "**Cast**". When you want to return to the old type you must only delete the contents of the string gadget and press <Enter>.

If you entered a wrong type the screen will flash and the former contents will be restored.

Changing Values

When you have selected a number variable (here for example the third variable "**illegal**"), you can directly modify the value of the variable in the "**Value**" string gadget, without having to bother with the hex editor.

When you selected a variable which type is a pointer to **UBYTE** or **BYTE** (also called a "string") you can change the text at string gadget "value". Pay attention not to enter more characters than is reserved for that string.

Sorting of Variables

First the list of variables is unsorted. Now you can select the sorting method with the cycle gadget "**Sort**". The alphabetical sorting will use the names of the variables for sorting, so you can find a certain one quicker in a big list. "**Last Changed**" will put the variable on the top that was changed last. This sorting will be updated after every program step. The debugger recognises every change, even changes of single struct fields or variables, which are changed through pointers.

The sorting method can be selected separately for every of the 3 variable pages.

Now execute some statements of the program by selecting the middle icon in the "**Debugger**" group. In the editor window you'll see how the program executes one "**printf**" function after the other, with the output appearing at the same time in the console window. The status line in the control window shows "**Breakpoint reached**" after each step. When you take a step that takes longer, for example when reading input from the console, you can also read the "**Program continues**" message.

At some point you'll reach the "**gets(s)**" statement on line 25. When you step through this function you will of course have to type the corresponding input in the console window.

Please enter <I> and then <Return>. In the switch-statement you'll now enter the first branch. On line 29 you'll find the function call for the function "**readAddressmask()**". This function resides in the "**address.c**" module. Don't select the step over button, but instead choose the second button from the left, to execute the program in single steps. This way you'll enter the function "**readAddressmask()**".

The source for the "**address.c**" module will now be loaded and the source code position will be shown. Now go step by step through the function. Depending on whether you use the second or third of the debugger icons in the control window, other function calls will either be stepped through or executed without stopping.

When needed, enter name, first name, street and place of residence in the console window.

Before ending the function, you can try the 4th button from the left in the "**debugger**" group. This one will execute the program without stopping until the end of the function is reached and you will return to the main program again.

Now you should be on line 30, and the variable "**address**" should have a sensible value. You can check this now: select the variable in the current variables window, and select the button "**I**" to examine it.

The elements of the "**struct Address**" will replace the variable list.



You can change the entered strings even now. Select e.g. the field **"name"** and change the contents in the string gadget of **"value"**. In this case you should only enter a string not longer than the old one, because the program has only reserved this space in the dynamic memory by **"malloc()"**.

Continue bug hunting and try a couple more things, for example watching variables and structure fields. When you feel comfortable with the debugger, end the program, either correctly or with **"Kill"**.

THE PROFILER

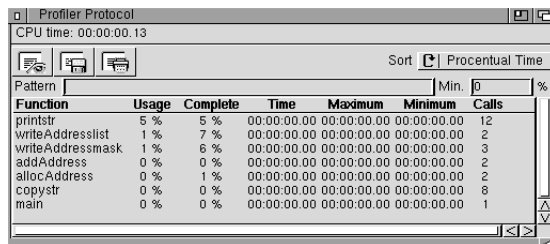
A profiler is an indispensable tool when optimizing a program. Compiler optimizations can only improve program performance by so much, whereas a profiler can provide the necessary information for identifying the most time-intensive functions in a program. These functions can then be rewritten to use better algorithms if possible, or at least sped up by carefully optimizing the source code by hand.

The StormC profiler is especially powerful; it allows precise timing and provides many valuable statistics about the program.

As always, we have stuck with the our maxim in that no special version of the program needs to be generated for using the profiler. Having the normal debug information generated will suffice. This - like the ability to start the profiler while debugging - is probably unique for compilers on the Amiga.

If you wish to use the profiler, make sure your project is compiled with the Debug option set to either “**small debug files**” or “**fat debug files**”. Select the “**Use profiler**” option in the Start Program window. The program can then be started as normal. Simultaneous debugging is possible, but may lead to minor deviations in the profiler’s timing measurements.

After starting the program, the profiler window can be opened.



The screenshot shows a window titled "Profiler Protocol" with a status bar indicating "CPU time: 00:00:00.13". The window contains a table with the following data:

Function	Usage	Complete	Time	Maximum	Minimum	Calls
printstr	5 %	5 %	00:00:00.00	00:00:00.00	00:00:00.00	12
writeAddresslist	1 %	7 %	00:00:00.00	00:00:00.00	00:00:00.00	2
writeAddressmask	1 %	6 %	00:00:00.00	00:00:00.00	00:00:00.00	3
addAddress	0 %	0 %	00:00:00.00	00:00:00.00	00:00:00.00	2
allocAddress	0 %	1 %	00:00:00.00	00:00:00.00	00:00:00.00	2
copystr	0 %	0 %	00:00:00.00	00:00:00.00	00:00:00.00	8
main	0 %	0 %	00:00:00.00	00:00:00.00	00:00:00.00	1

The upper-left corner updates the profiler display, changing all indicated percentage and timing values to reflect the latest results.



The help line shows the cumulated CPU time. This value is the amount of real CPU time used, ie. it does not include time that the program spends waiting (for signals, messages, or I/O) or time used by other programs that are running in the background.

The list below shows the functions including the following information:

Member functions of a class are displayed using the “**scope operator**” syntax (class name and member name separated by two colons).

1. *Function name.*

This counts only the time that the program spends in the function itself, or in OS functions called directly from it. Any time this function spends calling other functions in the program is omitted.

2. *Relative running time.*

This value provides the best hint as to which function uses up the most time. The sum of all values in this column will be 99 - 100% (the missing percent is lost in startup code and minute inaccuracies).

Here all subroutine calls from a function are included in its running time. For this reason the **main()** function will normally show a value of 99%.

3. *Relative recursive running time.*

These three lines give you a quick overview over the invocations of each function. Just how a function can be made faster often depends on whether the invocations generally take roughly the same amount of time to finish (small difference between longest and shortest running time), or some invocations take noticeably longer to complete than the others (great difference). In the latter case it may be profitable to optimize those special cases.

4. *Absolute running time.*
5. *Longest running time.*
6. *Shortest running time.*

Sometimes a function makes up a large chunk of the program's running time only because it is called very often, but each individual invocation takes up very little time. Optimizing such a function is usually a tough nut to crack. However it may be very beneficial in such a case to declare the function inline (“**__inline**” in C, “**inline**” in C++).

7. *Number of invocations.*

Above this list are several controls related to the profiler display:

The uppermost line is the help line which shows brief descriptions of the controls.

Directly below that, to the left, you see three buttons. The first of these updates the list of functions.

The second button lets you save the profiler display as an ASCII text file. A requester will appear to let you select a file name.

The third button dumps the information to the printer (using the "PRT:" device).

On the right-hand side of this line are the sorting controls for the function list. The first entry "**Relative**" sorts the list by the values in the second column, "**Recursive**" sorts by the third column, and "**Alphabetic**" sorts alphanumerically by the function name shown in the first column. And finally "**Calls**" sorts the list by the contents of the last column.

The line below this holds a text entry field for a DOS pattern string. Only those functions are shown whose names match the pattern; this can help reduce excessively long function lists to a manageable size. This can be used for instance to only show member functions of a particular class by entering the class name followed by "#?".

To the right of this sits a numeric entry field where you may enter the minimum percentage of running time that a function must take up in order to be shown in the list. Functions that make up less than 5 or 10% are often difficult to optimize and even doubling the speed of such a function is hardly worthwhile as the program would not become noticeably faster (a mere 2.5 or 5% in this case).

These optional restrictions aside, only those functions are ever shown that are invoked at least once while the program is running.

Double-clicking on a function entry will take you directly to its location in the source text.



The profiler display is also opened and updated automatically when the program terminates. The control window will also remain open. Closing the control window will also cause the profiler display to close and the list is forgotten. Should you want to have access to this information afterwards, make sure you have saved it to file or printed a hardcopy before closing the window.

Profiler technical information

The **LINE-\$A** instructions **\$A123** and **\$A124** are used to mark function calls.

These machine language instructions are unused on all members of the Motorola 68K processor family and trigger an exception. This exception is used to update the timing and call statistics.

The use of exceptions has the relative drawback of reducing the effective CPU speed, ie. the program will take longer to execute when the profiler is running than it does when the profiler is not activated. The difference will be particularly noticeable if the program invokes a lot of short functions. However the profiler will still be faster and more accurate than most existing profilers for the Amiga OS. The technique also buys the advantage of not having to recompile your code especially for profiling.

Handling of recursion is limited: The longest and shortest execution times will usually be unreliable, the total execution time (and therefore the relative values also) may be incorrect. A simple case of recursion (where **f()** calls **f()**) shows the correct relative values, but in the case of nested recursion (where eg. **f()** calls **g()** which calls **f()**) cumulates all times onto one of the two functions.

Function calls leading out of the recursion will still be shown correctly.

The effect on long jumps is not predictable, but in most cases this should only lead to minor distortions of the statistics for the called function.

Theoretically speaking, not all functions can be measured: Only functions whose machine code starts with a link or **movem** instruction are available to the profiler. One of these instructions will however be necessary in almost all cases,

even at the highest optimization levels. And fortunately any functions that do not need these instructions will be so small (no variables on the stack, only the registers **d0**, **d1**, **a0**, and **a1** are altered) that optimizing them any further would be near-impossible anyway.

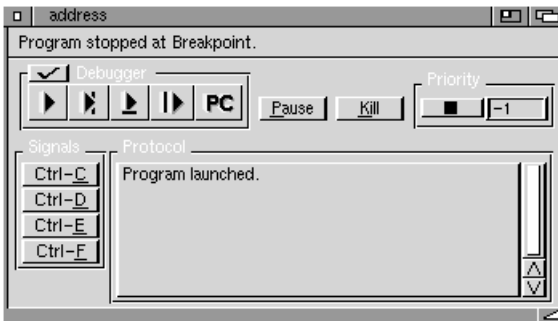
Inline functions generally cannot be measured.



REFERENCE

Control Window

The control window is used to direct the program and for resource-tracking. It is opened automatically at the start of a program.



Status line

Show short help texts about control window gadgets in text colour (black), and program status messages as important text (white).

"Program Stops At Breakpoint":

The program reached a breakpoint and stops. This can also be a breakpoint set by the debugger to make it possible to interrupt between C and C++ statements that often exists of more than one machine code command.

"Continue Program":

The program is running.

"Program waits for ...":

The program waits at an exec function "`wait()`" for signals. The signal mask is shown.

Debugger icons

The group with debugger icons contains a checkmark in the border to switch the debugger on and off. When the debugger is on the buttons that control execution are available.



When the program is started just „normal“ the debugger can be switched on at every time. Then the debug information is loaded. The program must be stopped by a suitable breakpoint. A module must be selected out of the module window to set the breakpoint in the corresponding source code or a function is selected to set the breakpoint there.

Go to next breakpoint



If you didn't put a breakpoint in your program, the program will be executed normally.

The program will execute at maximum CPU speed, until the next breakpoint is found (this needs to be set by the user in the editor). Because the debugger and the editor run in full multitasking together with the program that is being debugged, breakpoints can be set at any time.

The menu item **"Go"** in the debugger menu has the same effect.

Step in (single step)



This step will always go to the next breakpoint button visible in the source code. When present, function calls will be entered and breakpoint buttons will be used here similarly as well.

Functions without available source code will also be executed in single step mode, to allow stopping in function calls to code that has source code available again. Such jumps back can also happen in link libraries using function pointers or virtual function members in C++ classes. The execution speed of library functions is therefore low.

Functions of shared libraries will always be executed at full CPU speed. At recall of the program from such a library it can not be stopped. If the behaviour of a hook function should be tested a breakpoint must be set in the hook function before the call of the library function that calls the hook function.

Under certain circumstances it is critical to single step **"Forbid()"/"Permit()"** sections. The protection is broken after every stop (the program sends a message to the debugger and waits for an answer) and other tasks can run again. When at this time one of the global system lists is not correct the system might crash. When the **"Forbid()"/**



"**Permit()**" section is only used to copy the contents of a global data structure (e.g. the list of all waiting tasks of the system) it is probable that the result is incorrect.

Inline functions are normally treated like instructions, cause the compiler will insert inline functions directly into the program and optimise it very strong. So a allocation between instructions of the program and the source code of the inline function is impossible. For testing of a program teh compiler option "**Don't inline**" can be set and the compiler will treat every inline function like a normal one. So inline functions can be single stepped.

The menu item "**Step in**" in the "**Debugger**" menu has the same effect.

Step over (single step, but execute function calls without stopping)

This step will always go to the next breakpoint button visible in the source code, but function calls are executed at full CPU speed. If there is a breakpoint the program will naturally be halted.



The menu item "**Step over**" in the "**Debugger**" menu has the same effect.

Go to the end of the function.

The program will be single stepped until the current function returns to its caller. Function calls are executed at full speed.



The menu item "**Go to end of function**" in the "**Debugger**" menu has the same effect.

Show Current Program Position

This is the position after the last step or after reaching a breakpoint. When the source code position in th eeditor is lost, e.g. by viewing the source code position of a variable definition, it can be displayed by this button.



Normally the debugger displays all source sin the same editor window. At every change of it the old on will be removed out of memory and the new one is loaded. so the number of open windows can be minimised.

In cases a source is used very often it is sensible to keep it open permanently. So it must be opened by a double click on its entry in the project and not on the one in the module window. When the text is need the next time the breakpoints are automatically added. This way two or more source windows can be used.

The item **"Show PC"** of the **"Debugger"** menu has the same effect.

Pause

Stops the program while the button is lowered. The program will be removed from the list of running or waiting tasks. Processing of signals received while the program was paused is not guaranteed.

The menu item **"Pause"** in the **"Debugger"** menu has the same effect.

Kill

The program will be halted abruptly. No care is taken that the program isn't executing within a **"Forbid()"** - **"Permit()"**, or hasn't got a screen **"locked"**. **"Kill"** is therefore not safe under all circumstances.

The menu item **"Kill"** in the "Debugger" menu has the same effect.



Priority gadgets

The program priority can be altered from "-128" to "127", it is however recommended to leave the priority in the "-20" to "20" range, to avoid conflicts with certain OS tasks.

Normally the priority is set to "-1" so it is one time lower than the priority of the debugger and it will run perfectly.

Signal group

The signals **<Ctrl>-<C>**, **<Ctrl>-<D>**, **<Ctrl>-<E>** and **<Ctrl>-<F>** can be send at any moment. They are useful for simple program control while testing.

Protocol gadgets

The list itself shows all resources, for which no corresponding "free" statement was executed. By double clicking a line you can show the source code position of the



allocation, if this particular OS function is called directly in the program, i.e. if the functions are defined by "**#pragma amicall**" or "**#pragma tagcall**".

The screen is flashing when there is no corresponding source position for the resource. Typical resources without a corresponding source position are memory blocks and shared libraries. These resources are reserved by "storm.lib".

To avoid unneeded protocol output only these resource are displayed which reservation was made from the program. So resources which are directly reserved by a shared library are not displayed. Maybe one can imagine how many memory orders an "**OpenWindowTags()**" will cause...

Resources are freed very carefully. Commonly resources are pointers to a data structure and can a release by another OS function can not always be recorded. So there is a test at first, e.g. whether there is a screen with the given data structure "Screen" at the time it should be freed, before the screen will be closed by "**CloseScreen()**".

In most cases all resources can be freed.

The menu item "**Free resources**" in the "**Debugger**" menu frees all resources.

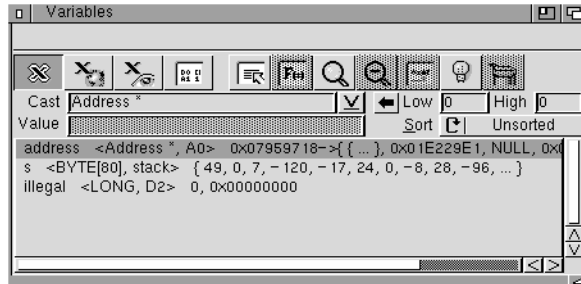
Window close gadget

The window will automatically be closed, when the program is ended and all resources are freed.

Closing the window automatically frees the resources.

Current variable window

The variable window shows all variables and inspected variables. It can be opened at any time by the item **"Variables..."** in the **"Windows"** menu.



Every variable is displayed with its name, type, memory class and value, e.g.

counter <LONG, stack> 42

The type is always displayed in the Amiga typical manner: **"LONG"** instead of **"long int"** and **"UBYTE"** instead of **"unsigned char"**.

The memory class rough marks the place where the variable is stored. The good optimisation of the compiler will often put variables to the same register or to temporary unable registers. So you should not wonder when a variable with the memory class **"D0"** will always change its value. This register is used very often.

List of memory classes:

"far data"

A global variable in the far data model.

"near data"

A global variable in the small data model with a4.

"near data (a6)"

A global variable in the small data model with a6.

"stack"

A local variable or a function parameter on the stack.

"D0" to "D7", "A0" to "A6", "FP0" to "FP7"

A variable in the relevant data, address or FPU register.



The values of these variables are displayed in the convenient C syntax. Many values can be interpreted differently so they will be displayed in more than one kind:

Integers are displayed decimal and hexadecimal. Signed types ("**LONG**", "**BYTE**" ...) are displayed decimal with sign and hexadecimal without it. "**UBYTE**" and "**BYTE**" (also "**unsigned char**" and "**char**") are interpreted as ANSI characters when there is no control character.

Floating points are only displayed in decimal. When they should be showed binary the hex editor must be used.

Counted types are displayed by their value and the name of the constant.

Pointers are displayed hexadecimal. The address is followed by an arrow "->" and the value at which the pointer shows. If the pointer shows to "**UBYTE**" or "**BYTE**" it will be interpreted as a pointer to a string and the first 40 characters will be displayed additionally.

Structures and arrays are displayed with parenthesis and the included elements. To limit the length of a value of a struct or array the numbers are viewed decimal, pointers only with the hexadecimal address and nested structures and arrays only with the parenthesis "{...}". Inspection allows a closer look at the values of these elements.

In the first line at the top short help texts will show the meaning of the icons. **Help**

Shows all variables and parameters of the current function and also the global variables of the module of this function.



Shows the global variables of all modules. Variables which are only defined "**extern**" are not displayed



Shows all watched variables.



This list can contain any variable or elements of an inspection. However the value of them is not always valid, e.g. register variables are only valid in certain areas of the function and variables which are on the stack are only valid as long as the program is in this function.



Shows the definition of the selected variables in the source code. The source code will be loaded and the cursor is positioned on the variable definition.



Displays the member function of the selected variables in a new window. In C++ only variables with a struct or class type can have member functions.



Inspects the selected variable. Therefore the current list of the variable (which can be an inspection itself) is replaced by a list of the elements of the variable.

If the variable has a struct, union or class type all members are displayed. If it has an array type the single fields of the array are showed. If it has a pointer type the type of this pointer is shown.



At a nested inspection the previously set inspection is displayed newly. Otherwise the variables that were shown before the inspection are displayed (current, global or viewed ones).



Opens the hex editor and positions the cursor on the address of the selected variable.



Puts the variable to the list of watched variables.



Deletes the variables from the list of watched variables.

Cast

Allows changing the type of a variable. The original type will still be shown in the type list of the window, the value however will be shown according to the new type.

By clearing the string gadget you can set it back to the original type.

The type entered must match the way the debugger writes them. Only simple types in Amiga specific form ("**BYTE**", "**LONG**", "**DOUBLE**"), pointers to these types ("**BYTE ***", "**LONG ***" etc.) and all types that are defined anywhere in the module are allowed.

Value

Numeric types allow direct modification without having to resort to the hex editor. The input can be decimal, octal or hexadecimal in the usual C syntax.



Strings can be changed too. That means a memory area on which a pointer on UBYTE or BYTE shows can be filled with a new string. It cannot be checked if the new string is too long, that means that the memory behind the reserved buffer of the old string will be overwritten (!).

The module window

The module window lists all the modules in the program. This window can be opened anytime using the menu item **"Modules"** in the **"Windows"** menu. By double clicking a module name in this list the source code for this module will be shown.



Modules without a debug file will not be displayed in the normal text colour (black) but in the selection colour (blue).

Loads the module source code into the editor.



Displays all functions of the module in a new window. Member functions are displayed with their qualified name, that means the name of the struct or class to which the function belongs will be displayed with two leading colons before the function name (e.g. **"String::left"**).

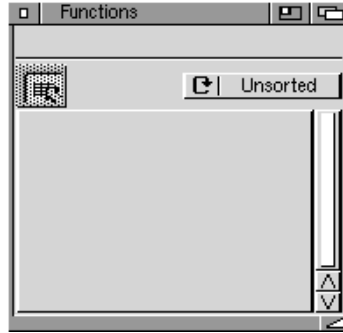


At the beginning the modules are displayed unsorted, that means they are in the order they appear in the project. With this option they can be sorted alphabetically.

Sorting

The function window

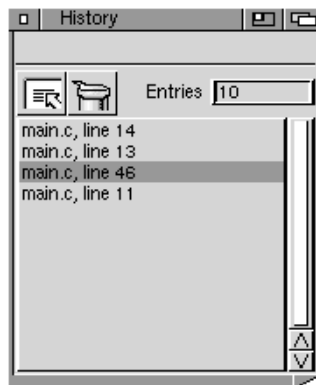
The function window is similar to the module window, but instead shows the functions in a module or the member function of a C++ class.



similarly a double click in this window will show the source code, only now the cursor is positioned at the start of the function. This is a comfortable way of setting breakpoints on functions.

The history window

The last actions of a program are recorded in the history window. So you can have a look at the course of your program in the source, e.g. when an unexpected behaviour appears and you do not know why.



The history window can also be opened by the item **"History..."** in the **"Windows"** menu.



A double click on an entry will display the source code in the editor.

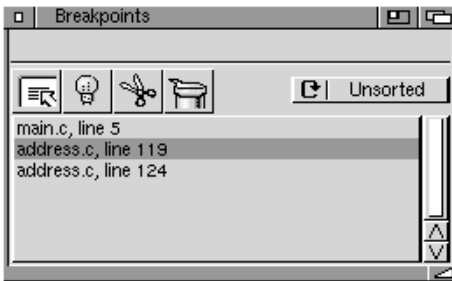
Displays the source in the editor.

Deletes all entries of the history. This should be done before watching a critical situation.

This sets the maximum number of entries in the history. The value must be between 10 and 256. **Entries**

The breakpoint window

All set breakpoints are displayed in this window. The window can also be opened by the time **"Breakpoints..."** in the **"Windows"** menu.



A double click will display the breakpoint sin the editor.

Displays the breakpoints in the editor.

Toggles breakpoints on or off.

Deletes a breakpoint.

Deletes all breakpoints.

Normally all breakpoints are displayed in order they were set. The sorting can be switched to alphabetical, where the first criterion is the module name and the second is the line number of the breakpoint. **Sorting**

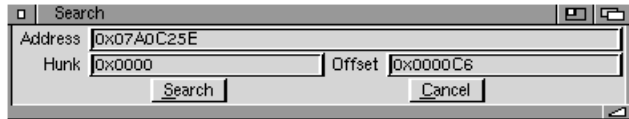
Pay attention on breakpoints that are in program parts that are used by different tasks simultaneous, e.g. a dispatcher of a BOOPSI gadget. Such a dispatcher is used by the program and by the task **"input.devide"** (if the gadget is part of an



open window). A breakpoint in the dispatcher could lead to an exception in the **"input.device"** and so cause a system crash.

The address requester

In the **"Debugger"** menu you can open an address requester using **"Find address"**. This shows the current addresses in the program in two forms and allows you to enter a new address in one of the two forms and find its position in the source code.



Address

This string gadget contains the address in its usual form.

Hunk

This string gadget contains the hunk of the program where the address resides.

Offset

This string gadget contains the offset relative to the hunk of the program where the address resides.

Search

Finds the source code position for the entered address, for example the hunk/offset pair.

Cancel

Closes the requester.

Every program on the Amiga consists of at least on hunk with number 0. Further hunks have increasing numbers. Every address that lies within a program corresponds exactly to a hunk and an offset, likewise every hunk/offset pair corresponds to an address, when you're searching one in a program.

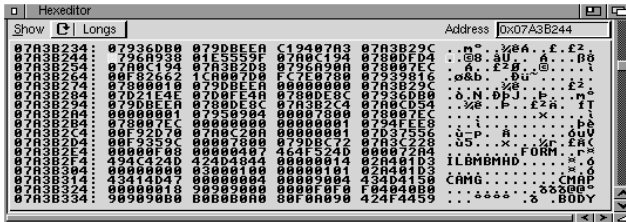
Many debugging-related programs use this form, since contrary to a physical address, the form hunk/offset is the same for every start of a program, whereas physical addresses may change each time.

As soon as you change any of the three values, an attempt is made to re-compute the other form. When you're looking for a particular hunk/offset pair (for example after your program caused an enforcer hit), this is an easy way to find

the physical address and subsequently the source code position using **"Search"**.

The hex editor

The hex editor shows an area of memory in hexadecimal and as ASCII characters. The display can be byte, word or long word based.



Choosing the display

This pop-up allows you to choose how to display the hexadecimal values. Byte, word and long word can be selected.

The address string gadget

This string gadget shows the current address under the cursor or allows you to set a new address.

The address column

In this column the address of the first byte of each line will be shown. These addresses can be changed directly as well, no need to switch to the address string gadget.

The hexadecimal column

This column displays the hexadecimal value the memory locations. If the address points to an illegal memory area then only minus characters are displayed and every modification is suppressed. The memory locations are not read to avoid enforcer-hits.

The ASCII column

This column displays the ANSI-character for the memory locations. If the value is outside the range of displayable ANSI characters a dot is shown, likewise for illegal memory addresses.

Keyboard control

The four cursor keys move the cursor within a column.

The <Tab> key switches to the next column.

In the address and hexadecimal columns the keys '0' to '9' and 'a' to 'f' (and these also with <Shift>, i.e. 'A' to 'F') are allowed.

In the ASCII column every character that is displayable one way or the other is of course allowed.

The scrollbar in the hex editor

To allow for comfortable operation of the scrollbar the total area covered by it is constrained to 8 Kbytes. However, if you move the mouse over the top or bottom border of the hex editor display the current address will be moved 4 Kbytes up or down, respectively. This allows you to also examine larger areas using the scrollbar.

For really big changes to the address it is always better to directly enter the address, because of the large address space of the CPU.



In a compiler system one rarely works with just a single source text. Splitting up a project in multiple parts becomes unavoidable after some time, be it to facilitate maintenance or because there is more than one programmer involved. An additional reason for dividing a project into several source files is higher compilation speed; recompilation after changing a single source file is faster as the other sources need not be processed again.

After compiling a multi-part project the linker is called upon, first, to combine the parts into a single program and second, to glue in the function libraries that high-level languages, particularly C and C++, tend to rely upon.

Two kinds of libraries exist in the Amiga system: Shared libraries, which are loaded at run-time and can be used by more than one program at the same time, and link libraries such as the standard ANSI library and the C++ class library which are attached (linked) directly to the program.

THE LINKER, THE UNKNOWN CREATURE	185
A first example	185
<i>ANSI-C Hello World</i>	<i>185</i>
Startup Code	186
Usage	189
Parameters	189
Memory classes	191
Compatibility	200
Error Messages	200
Error Messages	201
<i>Unknown symbol type</i>	<i>201</i>
<i>16-bit Data reloc out of range</i>	<i>201</i>
<i>8-bit data reloc out of range</i>	<i>201</i>
<i>Hunk type not Code/Data/BSS</i>	<i>201</i>
<i>16-bit code reloc out of range</i>	<i>201</i>
<i>8-bit code reloc out of range</i>	<i>201</i>
<i>Offset to data object is not 16-bit</i>	<i>201</i>
<i>Offset to code object is not 16-bit</i>	<i>202</i>
<i>Offset to data object is not 8-bit</i>	<i>202</i>
<i>Offset to code object is not 8-bit</i>	<i>202</i>

<i>Data- access to code</i>	202
<i>Code- access to data</i>	202
<i>InitModules() not used, but not empty</i>	202
<i>CleanupModules() not used, but not empty</i>	203
<i>File not found</i>	203
<i>Unknown number format</i>	203
<i>Symbol is not defined in this file</i>	203
<i>Nothing loaded, thus no linking</i>	203
<i>Can not write file</i>	203
<i>Program is already linked</i>	204
<i>Overlays not supported</i>	204
<i>Hunk is unknown</i>	204
<i>Program does not contain any code</i>	204
<i>Symbol not defined</i>	204
<i>Symbol renamed to _stub</i>	204
<i>_stub is undefined</i>	205
<i>32-Bit Reference for Symbol</i>	205
<i>32-bit Reloc to Data</i>	205
<i>32-bit Reloc to BSS</i>	205
<i>32-bit Reloc to Code</i>	205
<i>32 Bit Reference for symbol from FROMFILE to TOFILE</i>	205
<i>Jump chain across hunk > 32 KByte not possible</i>	205
<i>More than 32 KByte merged hunks</i>	205
<i>Illegal access to Linker-defined Symbol</i>	205
<i>Fatal errors : aborting</i>	206
<i>Hunk_lib inside Library</i>	206
<i>Hunk_Lib not found</i>	206
<i>Wrong type in Library</i>	206
Predefined values	206
<i>Symbols for data-data relocation</i>	208
<i>Hunk Layout</i>	208
<i>Memory Classes</i>	208
Order of Searching for Symbols	209
Hunk Order	209
Near Code / Near Data	209



THE LINKER, THE UNKNOWN CREATURE

The StormC package comes with an integrated linker called StormLink. This subprogram does a lot more than merely combine program and library modules, as will be explained below.

It is common practice in high-level languages to divide even relatively small projects over several modules. As mentioned in the introduction large benefits are to be had here; reduced compiler turnaround times and more manageable source files are only two examples.

Frequently-used routines may be combined into libraries and reused at will. These libraries may grow quite large, however the linker will only glue in the functions that are actually used, enabling wide standardisation and reuse of finished code without the need for translating it anew each time it is used.

A first example

Should you be unfamiliar with these concepts, allow me to explain them with an example. We shall use our well-known example program in ANSI-C:

ANSI-C Hello World

```
#include <stdio.h>
int main( void)
{
printf( "Hello World\n");
return 0;
}
```

The text output function used here is called **printf()**. Now if you create a project **ANSIWORLD.1**, and add only a file containing this little program (let's call it **ANSIWORLD.C**), you may compile and run it in the familiar way. The program will print "Hello World"; yet the **printf()** function has not been defined anywhere in your code.

So, where does this function come from and what does its code look like?

Actually the programmer need not concern himself with that. The function "**printf()**", with many others, belongs to the standard ANSI library that we have already programmed for you and included in the compiler system.

In most compiler systems, each program must eventually be run through a linker, which finds and includes any routines that the programmer may have used, to produce a working executable. How this is done internally isn't normally of any interest. What matters is how to make your wishes known to the linker, and how to configure the linking process to your needs.

Startup Code

Some internal tasks that need to be performed by any program upon startup, and some cleanup work when exiting, are handled for you by startup code that is included automatically by the linker. StormC comes with two varieties of this module called "**startup.o**" and "**library_startup.o**" (both can be found in the "**StormC:-StormSys**" drawer).

You will only need the "**library_startup.o**" startup code when programming shared libraries; it contains certain tables and other data structures necessary for initialising shared libraries such as the "**_SysBase**" variable, which the "**LibInit()**" function uses to keep the "**exec.library**" base pointer in. This module is used when "**Link as Shared Library**" and "**StormC Startup Code**" have been selected in the "**Linker 1**" page of the Linker Settings window.

The normal startup code is more interesting. It in turn comes in three flavours: One for the "large" data model, and two for the "small" data model (relative to a4 and to a6, respectively). However the linker will select the appropriate one of these three automatically.

All versions of the startup code detect whether your program is started from the CLI or from Workbench, and act accordingly. In the former case, the startup code first calls "**InitModules()**", then the function labeled "**main_**" (this is the parameterless version of "**main**" in C++) and finally "**CleanupModules()**".

In the later case, it first receives the Workbench startup message, then calls "**InitModules()**", and next the



function labeled **"wbmain_P09WBStartup"** (this is the version corresponding to **"wbmain(struct WBStartup*)"** in C++); finally it calls **"CleanupModules()"** and replies the Workbench startup message.

In order to support ANSI C, the **"storm.lib"** library contains standard functions that redirect the calls to **"main"**:

The standard **"main_"** function parses the CLI command-line arguments and stores them in an array which is then passed on to a function labeled **"main__iPPc"**, which is the C++ function **"main(int, char **)"**. If no such function exists, the ANSI-C **"main()"** function is called. This function at least must be defined; otherwise the linker will emit an error.

The standard function **"wbmain_P09WBStartup"** simply calls **"_wbmain"**, which is the ANSI-C function **"wbmain(struct WBStartup *)"**. Should there be no definition of this function, the program exits. As a result, starting a CLI-only program from the Workbench will exit quietly instead of causing a system crash.

These functions are all defined in the **"storm.lib"** library and may be overloaded if desired, for instance to make **"wbmain()"** print an error message if the program cannot be run from the Workbench.

The startup code module defines the following symbols:

The ANSI-C **"exit()"** function. This will call **"CleanupModules()"** and subsequently exits the program correctly. **_exit**

The **"abort()"** function used if no alternative has been configured using the **"signal()"** function. It exits the program without calling **"CleanupModules()"**. Exiting a program in this way makes little sense as libraries will not be closed and other resources, eg. allocated memory, will not be freed. The use of **"exit()"** is preferred. **abort__STANDARD**

This is where the **"exec.library"** base pointer is stored. It contains the same value found at the absolute address **"\$4"**, but is used to reduce the number of accesses to that memory area. **_SysBase**

Should you plan to write your own startup code, please take care to define these symbols if you want to use the **"storm.lib"** link library. The compiler itself expects certain functions in this library, for example those that perform 64-bit integer arithmetic, to be defined so you need to link with **"storm.lib"** if you intend to use that. However these functions do not depend on any of the above-mentioned symbols.

The small-data model versions will additionally copy the program's static data area to dynamic memory if the program is linked with the options **"Residentable Program"** and **"Write Reloc-table For Data To Data"** enabled. This will make the program residentable and (usually) re-entrant, so that it can be loaded into memory with the **"resident"** CLI command and run several times concurrently.

In addition to the possibility of defining your own startup code, you may also opt to use no startup code at all. If you choose to do this, you should include as a first source file in the project a small file containing only a minimal function that simply calls the real entry point of the program.

Example:

```
void pre_main(int, char *);

void my_startup(register __d0 int cmdlinelen,
               register __a0 char *cmdline)
{
    pre_main(cmdlinelen, cmdline);
}
```

The first source file in the project should contain no more than this.

The necessity for this really quite pointless function is that the order in which functions are defined in the object file need not match the order in which the source file defines them, and the executable file may put them in yet another order.

The only guarantee that StormC makes here is that the first file in a project will be the first one to be passed to the linker (provided of course that you link without startup code)

which will use the first code hunk in this file as the entry point of the executable.

Larger source files than the one shown above, with multiple functions and perhaps global variables (whether static or not) and include files could have the code for the functions in a different order than you may expect, eg. the **"INIT_"** and **"EXIT_"** functions for global variables in C++ are often put at the start of the object file. But even in ANSI-C the compiler will sometimes generate such **"INIT_"** functions, e.g. to initialise complex data structures.

Usage

The user normally doesn't see the linker, as it is called by StormC's integrated development environment. Started from CLI, StormLink will evaluate its command line and, if it contains only a single parameter **"AREXX"**, attempt to establish an ARExx message channel to a port called **"STORMSHELL"**.

Parameters

StormLink supports the following command-line options:

Selects the output file name.

TO

TO "filename"

Usage

StormLink does not check if a file of the same name already exists; if it does, it will be overwritten without asking. If more than one output file has been selected, the last one will be used.

Description

MAP

MAP

Usage

This option causes StormLink to create a file called a "linker map" containing all symbols in the created program and their place within the executable. Here is an extract from such a linker map:

Description

```
_cout = $ 24 in hunk 1 <stormc:lib/storm.lib> ( Far Public )
_std_out = $ 32 in hunk 1 <stormc:lib/storm.lib> ( Far Public )
_std_in = $ 16 in hunk 1 <stormc:lib/storm.lib> ( Far Public )
_cin = $ 8 in hunk 1 <stormc:lib/storm.lib> ( Far Public )
_clog = $ 40 in hunk 1 <stormc:lib/storm.lib> ( Far Public )
_std_err = $ 4E in hunk 1 <stormc:lib/storm.lib> ( Far Public )
_cerr = $ 40 in hunk 1 <stormc:lib/storm.lib> ( Far Public )
```



The first entry in every line contains the name of a symbol, followed by its offset in hexadecimal notation, and the number of the hunk in which it is defined; next, the name of the object file that the symbol originates from and finally its memory class.

LIB *Selects the linker's search path.*

Usage LIB | LIBRARY

Description Any object files specified after this option are opened under the names used on the command line; if they are not found, the linker looks for them in the "**StormC:LIB**" directory (this may be changed with the LIBPATH option). Failing this, it will see if that directory contains a file of the same library name but with ".lib" appended. Failing this, an error is reported.

LIBPATH
Usage LIBPATH

Description With this option you may configure where StormC looks for its library files. The default is "**StormC:LIB**".

CHIP *Sets the memory class for the finished program.*

Usage CHIP

Description Forces the entire program to be loaded into **Chip** memory.

FAST
Usage FAST

Description Forces the entire program to be loaded into Fast memory; this is the opposite to **CHIP**.

CAUTION! Attempting to run a program linked with the FAST option on an Amiga that has no Fast memory will result in an AmigaDOS error 103 (out of memory).



Memory classes

The Amiga architecture defines two types of RAM, to wit Chip and Fast memory. This distinction is made for reasons of efficiency.

Any data to be processed by the Amiga's custom chipset, eg. sound samples or sprite bitmaps must reside in Chip memory. This memory is shared between the processor and the chipset which accesses it continually, resulting in a significant slowdown for CPU access.

Depending on screen mode and DMA activity by e.g. the blitter, the processor may be restricted to use only as little as one fourth of the total bandwidth of this part of memory. For this reason most Amiga models come with additional so-called Fast-RAM, which is reserved for use by the CPU and which therefore sustains much higher access speeds. In addition it is usually run at a higher clock speed than Chip-RAM, increasing bandwidth even further. Program code should therefore preferably be loaded into Fast-RAM, if available; Chip memory should only be used when strictly necessary.

Fortunately the Amiga OS allows the programmer to specify what type of memory each program section (Glossary->Hunk) requires. This makes it possible to load a program into Chip memory specifically, should this become necessary. Usually however only small parts of the program need to reside in Chip-RAM. They can be compiled/assembled separately, or StormLink may be instructed to add the necessary directives to these parts of the program should they be missing in the object file. Note that StormLink cannot do this as selectively as the programmer could, using the **"#pragma chip"** directive. The following options can be used to make StormLink force all data, code or BSS sections into Chip or Fast memory respectively.

Please keep in mind that program code in Chip-RAM will run at no more than one-fourth the speed it would have in Fast-RAM.



CHIP

Enables CHIPCODE, CHIPDATA and CHIPBSS.

Usage

CHIPCODE CHIPDATA CHIPBSS

Description

These force the OS to load the indicated program sections into Chip-RAM.

For example, CHIPDATA may be needed when some graphical objects fail to appear on the screen. This option is to be used with caution as it causes all data to be loaded into Chip-RAM.

FAST

Enables FASTCODE, FASTDATA and FASTBSS.

Usage

FASTCODE FASTDATA FASTBSS

Description

These options are really only provided for sake of completeness, as using them forces the specified sections into Fast-RAM and the resulting program cannot be run if none is available. They may be useful in some cases though, as the OS will attempt to flush unused libraries from memory if necessary to free up Fast-RAM, rather than simply load the program into Chip-RAM. These options were included for no other reason.

ADDSYM

Create symbol hunks (HUNK_SYMBOL data)

Usage

ADDSYM

Description

This option copies symbol information from the object file into the output file, so that labels seen in a debugger will have meaningful names. StormLink also includes global symbols, so in some cases the name for a particular address may be included twice.

DEBUG

Usage

DEBUG

Description

Generates a ".LINK" file needed by the StormC system to run the program through the debugger.



ROMCODE CodeStart DataStart BssStart

ROMCODE

Usage

Description

For those who want to create EPROMs; use this option only if you really know what you're doing.

The output file will contain only the raw code and data hunks. The BSS hunk must be allocated manually at the appropriate absolute address using the "**AllLocAbs()**" function. Code and data hunks are relocated to the specified addresses at compile time, so there is no AmigaDOS relocation table overhead. After the indicated number of bytes of code, only the data section follows.

If your program addresses data relative to an address register, the BSS hunk is again assumed to be directly behind the data hunk. The BSS thus starts at Address of Data + Length of Data. The SmallCode, SmallData and SmallBSS are set automatically, so there is no fragmentation. Using the WARNREL option and corresponding startup code you may produce completely PC-relative code here.

Caution!

Linker database offset

BASE

BASE number

Usage

Sets the linker database's offset value within the program to First Data Object + number. Should your program contain too much data, resulting in a "16-Bit Reference/Reloc Out of Range" error from the linker, setting BASE to 0x8000 may help you out. If this doesn't work, you will need to switch to 32-bit addressing.

Description

Generates CODE hunk only

ONEHUNK

ONEHUNK

Usage

Produces a single CODE hunk containing all Code, Data and BSS sections. This could be useful for games programmers.

Description

Any relocation routine now only needs to deal with one hunk. If the program is compiled with the Small Data model, this allows completely PC-relative programming in a high-level language. The data base register (normally a4) is now initialised with

lea _LinkerDB(PC), a4

If all other addressing is done using **d16(pc)** or **d16(a4)**, the program will be entirely free of absolute addresses.

Such programs are fast in loading, but are normally non-reentrant. Combining this option with **ROMCODE** yields a program that can be loaded into some (SPECIFIC) absolute address using **Read(File, Buffer, len)**, and run directly. This technique requires that the program contains no absolute addresses whatsoever, meaning that the **WARNREL** option must be enabled if you want to apply it.

VERBOSE

Enables the multiple-definition check.

Usage

VERBOSE

Description

If **VERBOSE** is enabled, StormLink shows the names of any files loaded and scans them for multiply-defined symbols. Should one be found, it will print the names of the object files that define it.

The check also applies to libraries, as names can be used more than once there but with different meanings.

Perhaps the best example of this is the C **printf()** function: The "**MATH.LIB**" library defines a second version of this function, which is capable of printing floating-point numbers. The **VERBOSE** option is disabled for filenames following the **LIB** keyword, so that the program may still overload library symbols.

SMALLCODE, SMALLDATA, SMALLBSS

These options are superfluous under normal circumstances, as StormLink can independently decide what to do with a hunk. In some cases however this may not be what you intended, which is why these options allow you to influence how the linker combines the hunks.

These options take priority over and disable their respective **MAXCODE**, **MAXDATA** and **MAXBSS** counterparts.



Forces creation of only a single Code section.

SMALLCODE

SMALLDATA

SMALLBSS

Description

Specifying any of these options causes only a single hunk of the corresponding type to be generated. It will obey the strictest of the memory-class constraints for that hunk type.

Memory classes

See also

MAXCODE

MAXDATA

MAXBSS

Usage

MAXCODE number

MAXDATA number

MAXBSS number

StormLink normally merges all hunks of the same type, e.g. all **NEAR** hunks, all **PUBLIC-CODE** hunks, and so on, into one hunk for each type.

Description

Using these options you may fragment the program, i.e. create several smaller hunks of the same type. StormLink will merge as many hunks as will fit into the size specified.

Specifying **MAXCODE 1** will prevent any **CODE** hunks to be merged, so each one will appear as a separate hunk in the executable. Hunks marked **MERGED** are an exception; these will all be merged into a single hunk regardless.

RESIDENT

Usage

RES | RESIDENT

A program that is to be kept resident in memory makes some special requirements of the run-time system. Global variables must be kept for each instantiation separately so no other invocation of the same resident program can reach or alter them.

Description

A residentable program is essentially a program with the small data model that creates a new copy of its entire data section on each invocation and uses the copy exclusively. Above all it is vital that the program reference no absolute addresses. These would point to the original data section and



cannot be redirected to the copy. With the **RESIDENT** enabled a warning is emitted for such cases.

WARNREL

Warn of 32-bit relocations.

Usage

WARNREL

Description

Causes StormLink to emit a warning if the program contains 32-bit Relocs. This option is not to be confused with **RESIDENT**.

Use this option when creating position-independent code. In that case one should use the ONEHUNK option as well.

OPTIMIZE

Usage

OPTIMIZE

Description

When **OPTIMIZE** is enabled, StormLink makes an optimization pass over the object code. If a 32-bit reference to an object within the same hunk is found, StormLink will attempt to replace it by a PC-relative reference saving 4 bytes by freeing up a reloc entry.

This optimisation may result in incorrect code however. The optimiser contains a tiny "disassembler" that recognises only the **JSR**, **PEA** and **LEA** instructions. It cannot distinguish between code and any data that may be interspersed with it.

The optimiser is only called for global references, ie. when references are bound across object modules. Performing these optimisations for relocation entries is left to the assembler and the compiler.

NEARCODE

Description

StormLink assumes that all symbols needed for CleanupModules or InitModules can be reached by a single short jump. This may lead to problems with very large hunks, as this type of jump cannot cover a distance of more than 32 Kb.

OOP

Usage

OOP

Description

Generates the auto-init and auto-exit routines needed by a C++ compiler. When enabled, StormLink assembles a routine labeled **"_InitModules"** and one labeled



"**_CleanupModules**", which call the required "**_INIT_x_**" and "**_EXIT_x_**" routines.

Here "x" is a number between 0 and 9. In this case, and only in this one, the use of multiple symbols with the same name is allowed. If the routines are non-empty, yet never called, a warning is emitted.

The "**_InitModules**" routine calls the "**_INIT_0_**" routines first, and the "**_INIT_9_**" routines last. The order is reversed in "**_CleanupModules**". The entries are sorted by number, where single-digit numbers are multiplied by 100 and two-digit numbers by 10. Thus "**_INIT_1_**" is called before "**_INIT_31_**", ensuring compatibility with the three-digit priority numbers in the SAS/C compiler.

LOG

LOG "filename"

Usage

Whenever a symbol is referenced that StormLink can't find, an ERROR 25 message is generated. Before this happens, however, it tries to find out which file it could load to find the symbol anyway.

Description

The LOG option loads a log file, which contains information about object files and the symbols defined in them. It looks like this:

```
<filename> <= This is the name of the object file
<symbol> <= Symbols
<symbol>
```

The filenames start at the first column, whereas symbols are recognised by leading whitespace.

DDREL

DDREL

Usage

The abbreviation means Data-Data Relocation. If there are any fixed 32-bit relocations from Data to Data/BSS, this option instructs StormLink to set up an internal relocation table for the program so that the startup code can work out the relocations for its own copy of the Data/BSS hunks. This option is useful only in conjunction with **RESIDENT**.

Description



FORCELINK

Usage

FORCELINK

Description

This option forces StormLink to include all Data and BSS hunks from all object files included in the command line.

Unused hunks from libraries are still stripped from the output file. Using this option may become necessary if your program contains global classes with constructors that perform useful work although the objects of those classes are never used in the program. StormLink would attempt to discard these objects as unused, which may not be what you intended.

MODEL

Usage

MODEL [FAR - NEAR - NEAR_A6]

Description

To simplify library handling, StormLink allows you to ignore certain parts of a library.

When, for instance, the MODEL option is used with the FAR argument all parts compiled with the **NEAR** or **NEAR_A6** options are omitted. As a result a library may contain multiple versions of each of its symbols, obviating the need for a plethora of different versions of each library. With this option the linker can be instructed to simply select the desired version of each function.

FOLDCODE

Usage

FOLDCODE

Description

StormLink is able to perform several optimisations on the final executable similar to those performed by a compiler. One of these is detection and removal of duplicated code, which is enabled by this option.

When StormLink finds two identical routines, where one could be replaced by the other without affecting the result, the second routine is removed and replaced by the first one.

The use of this option may not seem apparent; after all you know what is in your program and wouldn't define identical functions. The answer is in C++ template instantiations. An executable may contain many identical instantiations of a function template or class template member function



resulting in potentially enormous code bloat and deteriorated cache performance.

As this optimisation may take some time to perform its use is only recommended when compiling at optimisation level 9, and you have either a fast machine or enough time to spare.

VERSION

VERSION "number"

Usage

Sets the global **constant** `_ _VERSION` to the numerical value of "number".

Description

REVISION

REVISION "number"

Usage

Sets the global **constant** `_ _REVISION` to the numerical value of "number".

Description

ARexx StormLink has an ARexx port called STORMLINK. Although full documentation of this port and its commands could be of interest to some users, it is too large for the scope of this manual section.

This port is not intended for use from Rexx. Suffice it to say that StormLink, when called from the CLI without arguments goes into ARexx mode. Four commands are supported, each of which may however give rise to a flood of communications with StormC's IDE.

CD

CD "name"

Usage

Sets StormLink's working directory to the directory named by the argument.

Description

LINK

LINK files.o Options...

Usage

Initiates the linking stage with the specified options and object files.

Description

BREAK

BREAK

Usage



Description	Interrupts an ongoing linking stage, if possible.
QUIT	
Usage	QUIT
Description	Remove StormLink from memory

Compatibility

StormLink is highly compatible with BLink and with SLink from the SAS/C package. StormLink supports both the SAS/C library format as the SAS/C constructor and destructor features. Combined use with StormC poses no problems, so you may use them together.

Error Messages

One design goal for StormLink was proper behaviour under error conditions. All error messages consist of several parts: First comes either the word "**warning**" or the word "**Error**", followed by the number of the error message. Then comes a brief explanation with some parameters similar in appearance to a command line. This enables you to quickly find the necessary information.

The following message parameters may occur:

TYPE	This keyword is followed by the type of the hunk in which the error occurred, ie. "Code", "Data" or "BSS".
FILE	Indicates the object file in which the error occurs.
FROMFILE TOFILE	As the job of the linker is to combine object files, some errors are caused by more than one file. In such cases the FROMFILE keyword indicates the file containing the problematic reference and TOFILE shows the object file it points to.
SYMBOL	Shows the name of the symbol involved in the error.
OFFSET	Shows the file offset at which the problem occurred.
PREFILE	This parameter is printed only in the case of a multiply defined symbol; it shows the name of the file containing its first definition.



Followed by a number, this parameter indicates an invalid object file.

HUNKTYPE

Most of these errors are not fatal and will not cause the linking stage to be aborted. They are, however, of importance with regard to the produced executable as they indicate that something has gone wrong at the place indicated in the error message.

Running such an executable will most likely result in a system crash ! If you are certain that the symbol involved is never used then doing so is nevertheless still possible, e.g. in the debugger.

Error Messages***Unknown symbol type***

This is a symptom of an incorrect object file; it contains a symbol with a type designator that StormLink doesn't know. *Error 0.*

16-bit Data reloc out of range

This error can occur when SmallData hunks get too large. If BASE is already set to its maximum (0x8000) then you have a problem. The program contains too many data for the small data model. *Error 1.*

8-bit data reloc out of range

See error 1; this error has the same cause but should be more rare. *Error 2.*

Hunk type not Code/Data/BSS

This object file is corrupted, if it is an object file at all. *Error 3.*

16-bit code reloc out of range

Similar to error 1, but for code rather than for data. StormLink encountered a 16-bit reference spanning too long a distance. *Error 4.*

8-bit code reloc out of range

See above. *Error 5.*

Offset to data object is not 16-bit

The distance to an imported data object is too large. See error 1. *Error 6.*

Offset to code object is not 16-bit

Error 7.

This error should never occur. If such a situation occurs, StormLink will build a chain of short jumps to span the same distance. Only if even the jump out of the hunk is too long can this still be a problem.

Offset to data object is not 8-bit

Error 8.

This error is analogous to error 6 with the difference that the maximum allowed distance is limited to only 8 bits.

Solution

Find the corresponding symbol in the map file and set the BASE value manually (between -127 and +127). If errors of this kind still occur, getting the program to link is next to impossible. The only solution is to change the source code causing the problem.

Offset to code object is not 8-bit

Error 9.

The usual cause for this error are object files created by an assembler that contain a directive of the form "**bsr.s _external_name**". This error should not occur with object files created by StormC, as it doesn't (and mustn't) use such constructs. The indicated object file apparently contains a reference to a code object that cannot be resolved.

Solution

Rewrite the source code causing the problem.

Data- access to code

Error 10.

Object files can state whether a reference should point to a data object or into the code area. If StormLink discovers a violation of this restriction, it rings the alarm. This error is caused by such constructs as "**extern long printf; printf=0;**", which is absolutely fatal to the program! The VERBOSE option enables this warning.

Code- access to data

Error 11.

This error is currently disabled as it would give false alarms continually. I have yet to find an assembler that can generate the necessary data references and does not use the same type of reference for code and data all the time.

InitModules() not used, but not empty

Error 12.

All modules that need any kind of initialisation export a so-called constructor, and a destructor for cleaning up afterwards.



StormLink automatically generates code to call these routines. The program first calls the InitModules routine, usually from the "**STARTUP.0**" startup code, which in turn calls all the constructors. If this is not done, ie. if the routine is never called yet does contain code, the program cannot be correct unless it would call all **INIT_xx** and **EXIT_xx** routines by itself.

Seeing that the linker knows best just what is in the list of necessary routines and what isn't, you shouldn't even try to do this by hand. In strict accordance to Murphy's Law, one will always be forgotten.

CleanupModules() not used, but not empty

See error 12.

Error 13.

File not found

The problem is obvious: StormLink can't find any file with the name you specify. *Error 14.*

Unknown number format

StormLink has its own parser to recognise and process different numeric notations such as decimal, octal and hexadecimal numbers in two different formats. This parser is also smaller than its "**STORM.LIB**" equivalent. Hexadecimal numbers may begin with either "**0x**" or "**\$**". This error message is displayed when a number contains a character that doesn't belong there. *Error 15.*

Symbol is not defined in this file

StormLink is also capable of reading index files from the MaxonC compiler, which calls them "**LOGFILE**". If this file should state that some symbol is defined in a particular object file that hasn't been loaded yet, but no definition for it is found in that file, this error is displayed. *Error 16.*

Correct the "**LOGFILE**" file.

Solution

Nothing loaded, thus no linking

The command-line arguments contain only options, and no object files. Therefore StormLink can't create an executable. *Error 17.*

Can not write file

For some reason this output file couldn't be written to. *Error 19.*

Redefinition of symbol in file FILE , first defined in file PREFILE

Warning 20.

StormLink has found a new definition for a symbol that has already been defined. As this can lead to errors that are difficult to track down, it is advisable to rectify this as soon as possible.

Program is already linked

Error 21.

Once a file has been linked, it no longer contains most of the information needed by a linker to operate on it. It is almost impossible to alter an already linked file in a meaningful way. For this reason StormLink refuses to load such a file at all.

Overlays not supported

Error 22.

There is nothing StormLink can do with an object file that requests an overlay. In this day and age when all computers have megabytes of memory, overlays are no longer of any use. Please use a shared library instead.

Hunk is unknown

Error 23.

There appears to be something wrong with this object file. The field that was expected to contain a valid hunk-type designator contains nothing that the linker can recognise.

Program does not contain any code

Error 24.

After stripping any unused parts of the program, the linker was left with a file that contained no code at all. This program can't do anything.

Symbol not defined

Error 25.

A symbol is used for which no definition can be found. It was either misspelled or is missing from the used libraries. If the message has a HINT parameter attached it points at the last symbol defined before the error, which is usually the routine that contains the unresolved reference.

Symbol renamed to _stub

Warning 27.

Any references to undefined symbols are automatically diverted to a routine called "**stub()**". The reason for this is that you may sometimes want to compile large projects before all routines have been written.



_stub is undefined

The "**stub()**" routine simply doesn't exist.

Error 28.

32-Bit Reference for Symbol

This message is enabled by the **WARNREL** option. It tells you that a 32-bit reference exists for a symbol that shouldn't be there because of the **WARNREL** option.

Warning 29.

32-bit Reloc to Data

This warning is also enabled by the **WARNREL** option.

Warning 31.

32-bit Reloc to BSS

Like warnings 29 and 31 above, this warning is enabled by the **WARNREL** option.

Warning 32.

32-bit Reloc to Code

Like warnings 29, 31 and 32 above this warning is enabled by the **WARNREL** option.

Warning 33.

32 Bit Reference for symbol from FROMFILE to TOFILE

This warning gives the filenames connected by the 32-bit reference to this symbol. The **WARNREL** option tells StormLink it should emit a warning for this case.

Warning 34.

Jump chain across hunk > 32 KByte not possible

A jump chain needs to hop over a hunk that is too large for a single jump.

Error 35.

Compile the module that causes the problem with the "**Create Library**" option.

Solution

More than 32 KByte merged hunks

A MERGED hunk uses 16-bit addressing, which is limited to a range of +/- 32 Kb (32768 bytes). This could, but need not, lead to problems.

Warning 36.

Illegal access to Linker-defined Symbol

StormLink defines a few symbols of its own for specific tasks. Should any such symbol be erroneously accessed by the user program, this error message is displayed.

Error 37.

Fatal errors : aborting

Error 38.

This indicates the occurrence of such serious problems while linking that StormLink can't handle them anymore. Trying to proceed under these conditions would only cause more problems, which the linker prevents by aborting the process. No output file is generated.

Hunk_lib inside Library ??

Error 39.

A file that has already been identified as a SAS library contains another such identifier within the library contents. StormLink doesn't understand this and exits.

Hunk_Lib not found

Error 40.

An important part of a file that has been identified as a SAS library is missing. This indicates a corrupted library.

Wrong type in Library

Error 41.

Symbols are represented differently in SAS library than they are in standard object files. Types simply added in the normal way confuse StormLink.

Predefined values

StormLink defines constant symbols for the linker database and for the length of the Data and BSS hunks. Notation is compatible with that used by BLink 6.7, however BLink will not show all of these symbols.

_LinkerDB

Usage

_LinkerDB: (label)

Description

Points to the first data element + **BASE**. With **BASE** set to 0 (this is the default) this points to the first element of the Near Data hunk. If the **SMALLDATA** option is used there may well be something before this.

_SmallData

Usage

_SmallData: (label)

Description

Points to the first data element + **0x8000**; this is used by the MaxonC++ compiler which assumes that **BASE** is always set to its maximum (**0x8000**). Mixing object files that assume different **BASE** settings is possible, but the produced executable will crash inevitably.



RESIDENT

RESIDENT: (const) 1

Usage

set when linking with **RESIDENT** option; 0 otherwise.

Description

__VERSION

__VERSION: (const) variable.

Usage

This is equal to 0 by default, but can be set with the **VERSION** option.

Description

__REVISION

__REVISION: (const) variable.

Usage

This is equal to 0 by default, but can be set with the **REVISION** option.

Description

SmallData Symbols

__DATALEN: (const)

Usage

Length of the data area in longwords (one longword equals four bytes) of the **MERGED** hunk.

Description

__BSSBAS (label)

Usage

Start of the BSS section in the **MERGED** hunk.

Description

__BSSLLEN: (const)

Usage

Length of the BSS section of the **MERGED** hunk, in longwords.

Description

__OFFSET: (const)

Usage

This equals the offset defined by **BASE** + the amount of data placed before the Near Data section of the Data hunk.

Description

These values are useful when writing a pure-resident program. It also allows the startup code to determine whether the BSS section still needs to be initialised or whether the program has been linked with the **RESIDENT** option in which case it must set up a new **MERGED** hunk. For more detailed information please read the provided source to the startup code.

Symbols for data-data relocation

`_DDSIZE`: (const) Length of table (in bytes!)

Description

Had this length been measured in longwords, the following would no longer work.

_DDTABLE: (label)

Description

Start address of table. This table is attached to the first Code hunk. If there is only one code hunk, (`_DDTABLE+_DDSIZE`) points to the end of the hunk. If no **DDREL**-option has been set, `_DDSIZE` equals zero.

Hunk Layout

Aggressive hunk merging is pursued to make the generated executables as small as possible.

Memory Classes

For the purpose of hunk merging, StormLink assigns a signature to each hunk. This signature contains:

- Hunk type (Code, Data or Bss)
- Memory Class (Public, Chip or Fast)
- Addressing Mode (Near or Far)

Naturally one may also use hunks that have already been marked with the Chip or the Fast bit. In that case, priorities are as follows:

CHIP (maximum)

FAST

PUBLIC (minimum, regardless of memory class)

Setting the Fast bit isn't very polite as it requires the presence of Fast-RAM in the system. CHIP data should be declared as Far if at all possible.

Assembler programmers should address CHIP data with 32-bit addressing only. This allows StormLink to merge such data into a single **CHIP-DATA** hunk. This way the Small Data hunk is free to be allocated in expansion RAM. Using so much as a single 16-bit access to a **CHIP** data element



through the **xx(ax)** addressing mode will force the entire Near Data hunk into Chip-RAM.

Order of Searching for Symbols

Symbols are looked for first in the Code section, then in the Data section and finally in the BSS section. If any symbols are not found, the log file is inspected to find out which object file defines them.

Hunk Order

All hunks of the same signature are merged together as far as any used options such as **SMALLCODE** or **MAXDATA** will allow. As StormLink can and will reorder hunks while linking, one should not rely on hunks appearing in the executable in the same order as they do in the object file. The linker first writes all Code sections, then all Data and finally the BSS sections. However the order in which StormLink processes its parameters is significant. It first makes a pass over the command line to filter out the options, and then attempts to load all other arguments as object files.

Near Code / Near Data

StormLink supports programming in the Near code model where routines are accessed through PC-relative addressing to reduce both execution time and program size.

Near Data consists of a hunk containing all pre-initialised data of the program as well as all un-initialised data (**BSS**). This hunk is addressed in a base-relative fashion, i.e. relative to a base address register. These accesses also have the speed and size advantage over 32-bit absolute addressing. Should the Near Code hunk grow larger than 32 Kb, StormLink constructs jump chains across single hunks to allow calls to span distances larger than 32 kilobytes.

Please take due care when using global data contained in the **CODE** section; the linker has no way of telling such data apart from program code. It may insert intermediate jumps where a data access was required, which could lead to bugs that are extremely hard to find.





Symbols

##base 125
##bias 125
##end 126
##private 126
##public 126
#define 72
#pragma header 71
**** 114
__COMPMODE__ 115
__cplusplus 115
__DATE__ 115
__FILE__ 115
__FUNC__ 116
__inline 109
__LINE__ 116
__STDC__ 116
__STORM__ 116
__TIME__ 116
__LibNameString 128
__LibVersionString 128
__stub 204
__wmain 187

A

amiga.lib 118, 156
Authors 2
auto 107
AutoSave 52

C

CleanupModules 113, 186
Compiler
 CLI version 130
 Options 134
 Special features 107
Compiler Options
 ANSI C or C++ 137
 Assembler source 136
 Code model 138
 Colours and styles 142
 Core memories 144

Data model 138
 Definition of symbols 136
 Error file 142
 Exception handling 137
 Include files 137
 Linker libraries 141
 Optimisations 138
 Optional warnings 143
 Pre-compiled header files 144
 RunShell 141
 Special processors 140
 Summary 145
 Symbolic debugger 141
 Template functions 138
 Treat warnings like errors 144
 Warnings and errors 141

Constructors 113

Copyrights 2

D

Debug Files 76

Debugger 76

Address requester 180
 Breakpoint window 179
 Changing Values 161
 Control Window 169
 Current variable window 174
 Display member function 176
 Don't inline 171
 Example 157
 Free Resources 156
 Free resources 173
 Freeze the program temporarily 153
 Function window 178
 General information 151
 Global variables 160
 Halting the program 154
 Hex editor 181
 Keyboard control 182
 Icons 169
 Inspect Variables 176
 Inspection 160
 Kill 172
 Local variables 160
 Module window 177
 Pause 172
 Priority 172
 Protocol 172

Resource-Tracking 152
Sending signals 154
Signals 172
Single step 170
Sorting of Variables 161
Task priority 154
Temporary Casts 161
Variable Window 159
Watched variables 160

Destructor 113

E

Exceptions 75

EXIT_ 113, 117, 128, 189

extern 107

F

FD files 125

H

Header Files 71

Hello World 185

history window 178

Homepage 8

Hotline 8

I

INIT_ 113, 117, 127, 189

InitModules 113, 186

Inline 108

Internet 7

J

Joining Lines 114

K

Kill 154, 163

L

LibClose 126

LibExpunge 126

LibInit 126, 186

LibNull 126

LibOpen 126

library_startup.o 186

Licensee 3

Linker

 _exit 187
 Compatibility 200
 Error Messages 200
 First example 185
 Hunk Layout 208
 Hunk Order 209
 Memory Classes 208
 Memory classes 191
 Near Code 209
 Near Dat 209
 Predefined values 206
 __BSSBAS 207
 __BSSLEN 207
 __DATALEN 207
 __OFFSET 207
 __REVISION 207
 __SmallData 206
 __VERSION 207
 __DDTABLE 208
 __LinkerDB 206
 RESIDENT 207
 Startup Code 186

Linker parameters

 ADDSYM 192
 BASE 193
 BREAK 199
 CD 199
 CHIP 190, 192
 DDREL 197
 DEBUG 192
 FAST 190, 192
 FOLDPCODE 198
 FORCELINK 198
 LIB 190
 LIBPATH 190
 LINK 199
 LOG 197
 MAP 189
 MAXBSS 195
 MAXCODE 195
 MAXDATA 195
 MODEL 198
 NEARCODE 196



ONEHUNK 193
 OOP 196
 OPTIMIZE 196
 QUIT 200
 RESIDENT 195
 REVISION 199
 ROMCODE 193
 SMALLBSS 195
 SMALLCODE 195
 SMALLDATA 195
 TO 189
 VERBOSE 194
 VERSION 199
 WARNREL 196

M

main_ 186
main_ippc 187

N

Nested Comments 73

O

Optimiser 84
Optimising 78

P

Phone 8
pragma - 110
pragma + 110
pragma amicall 111, 118, 156, 173
pragma chip 111, 191
pragma fast 111
pragma header 71
Pragma instructions 110
pragma libbase 124
pragma priority 112
pragma tagcall 111, 156, 173
Predefined symbols 115
Preface 5
Problems 7
Profiler 164
protocol list 155

Prototypes 117

R

register 107
Register parameters 108
Resource-Tracking 151
RunShell 151

S

ScreenManager 48
Settings 51
Shared Libraries

avail flush 129
 FD files 125
 Important hints 128
 Initialisation 127
 library_startup.o 128
 Make your own 123
 Project Settings 125
 Register Setup 124
 Setup 123
 Using 117

Source Level Debugger 151

startup.o 186

static 107

storm.lib 113, 127, 187

StormLink 185

STORMPRAGMAS 73, 118

Stub Functions 118

Support 8

T

Technical support 7

Textoptions 95

Thanks 6

Tooltypes 47, 94

GOLDED 48
 HOTHHELP 48
 PUBSCREEN 48
 QUIET 48
 SAVEMEM 48



W

Warnings 80

wbmain_P09WBStartup 187

WWW 8